FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Paralelização de Algoritmos de Processamento de Imagem em Contexto de Vinha

Filipe Alberto Sampaio Azevedo

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Orientador: Hélio Mendes de Sousa Mendonça

Co-orientador: Filipe Baptista Neves dos Santos

26 de Julho de 2018

Resumo

O crescimento da população mundial e a necessidade de haver uma agricultura eficiente e boa para o ambiente, exige uma agricultura de precisão, inteligente e com processos mais automatizados. Concretamente, a agricultura de vinha de encosta, devido à topologia do terreno, requer um sistema de localização preciso e eficaz, nem sempre disponível a tempo inteiro quando se utiliza um sistema unicamente baseado em *Global Navigation Satellite System* (GNSS).

Assim, um dos grandes desafios atuais da robótica para agricultura é alcançar algoritmos de processamento de imagem que sejam robustos às mais variadas condições topológicas, luminosas e temporais, e que, satisfazendo os requisitos do sistema robótico, em tempo real, apresentem custos ajustados à realidade da aplicação. Surgiu, portanto, o ViTruDe (*Vine Trunks and Masts Detector*), um algoritmo de perceção visual semântica para deteção de elementos naturais nas vinhas, nomeadamente os mastros e troncos, para auxílio à localização do AgRob V16, uma plataforma robótica que está a ser desenvolvida de forma modular para a execução de tarefas relacionadas com agricultura de encosta, como monitorização, pulverização, poda e colheita. Este *robot* de pequeno/médio porte foi então totalmente desenhado e construído para operar neste contexto de forma completamente autónoma, robusta e em segurança.

O ViTruDe necessita de ser executado com características de tempo real, no entanto, a sua implementação original em *Central Processing Units* (CPU) está longe de cumprir os requisitos mínimos de uma execução desse tipo. Desta forma, nesta dissertação, o algoritmo foi reprogramado para tirar partido da capacidade de paralelização dos *Graphics Processing Units* (GPUs) utilizando o modelo de programação paralela da NVIDIA, o CUDA. O ViTruDe foi então analisado e separado em segmentos mais pequenos e simples, que pudessem ser distribuídos por diferentes núcleos de uma placa gráfica e desta maneira correr concorrentemente. Isto permitiu melhorar a velocidade da computação e assim contribuir para um melhor desempenho em tempo real da plataforma robótica.

Os resultados obtidos, relativamente ao tempo de processamento, demonstram que esta implementação em GPU permitiu alcançar uma solução cerca de 80 vezes mais eficiente do que a abordagem original em CPU. O algoritmo fica assim com um tempo de execução dentro do que era desejado para este sistema de processamento de imagem. Todos os dados de treino e de teste utilizados nesta dissertação foram disponibilizados publicamente para possível utilização em trabalhos futuros.

Palavras-Chave: Aceleração de Processamento de Imagem, Agricultura, CUDA, GPU, Localização, NVIDIA, Robótica, Vinhas.

Abstract

The world's population growth and the necessity of an environment friendly and efficient agriculture, requires an agricultural sector which is precise, smart and with its tasks and processes more and more automated. Specifically, slope vineyard agriculture, due to the terrain characteristics, needs an accurate and efficient localization system, which isn't available at all times when using solely a Global Navigation Satellite System (GNSSS).

Therefore, one of the big challenges of agriculture robotics is to achieve real-time image processing algorithms which are robust to all kinds of topological, luminous and temporal conditions, while also being time efficient and with an adequate cost. Between this and that, what came to light was ViTruDe (Vine Trunks and Masts Detector), a visual perception algorithm to detect a high density of vineyard natural elements, like masts and trunks. This algorithm aims to help a robot acknowledge his location. This aforementioned robot is AgRob V16, that is being developed in a modular fashion to execute slope vineyards specific tasks, such as monitoring, spraying, pruning and harvesting. This small/medium-sized robot was completely designed to operate autonomously and safely.

ViTruDe needs a real-time performance that is simply not allowed by its original Central Processing Units (CPU) implementation. This way, in this dissertation, the algorithm was reprogrammed to take advantage of Graphics Processing Units (GPU) parallelization capability with NVIDIA's CUDA programming model. ViTruDe was analyzed and split into smaller, simpler segments of code in order for them to be programmed and distributed among different graphical card cores, running simultaneously. This allowed a significant increase in the computational speed and thus contributed to a better real-time performance of the robotic platform.

The obtained results show that, for this GPU implementation, the algorithm is over 80 times more efficient than its CPU approach regarding total execution time. All training and test data used were made public for possible use in future works.

Keywords: Agriculture, CUDA, GPU, Image Processing Acceleration, Localization, NVIDIA, Robotics, Vineyards.

Agradecimentos

A realização desta dissertação foi, tal como qualquer outra, um percurso longo e fruto de muita dedicação e trabalho.

Primeiramente, quero agradecer ao meu orientador, o Professor Doutor Hélio Mendes de Sousa Mendonça, e ao meu coorientador, Doutor Filipe Baptista Neves dos Santos, porque na sua ausência esta dissertação não existia. Desde o primeiro dia, quando eu estava ainda à procura de um tema de tese que me enchesse as medidas, o Dr. Filipe convenceu-me a participar neste projeto e a optar por fazer algo que eu nunca tinha feito. Desafiou-me a sair da minha zona de conforto e a realizar uma dissertação numa área completamente nova para mim, assegurando-se de que os trabalhos estavam bem encaminhados. Eu pude sempre contar com os seus conselhos e a sua preciosa ajuda, que desbloqueou as tarefas que pareciam mais complicadas. Por isto tudo, quero deixar-lhe um enorme obrigado!

Quero agradecer também ao Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência (INESC TEC) e a todo o pessoal do laboratório do Centro de Robótica Industrial e Sistemas Inteligentes (CRIIS), onde eu pude desenvolver a maior parte dos trabalhos, em especial ao Luís Santos e ao Jorge Mendes, pela disponibilidade em ajudar sempre que foi necessário.

Aproveito esta secção para agradecer também aos meus amigos, que nunca me deixam abdicar da vida social, e à minha namorada, pelo seu apoio e por ter revisto esta dissertação inúmeras vezes para me dar conta dos erros pontuais e gramaticais que estão sempre presentes.

Por último, mas definitivamente não menos importante, quero agradecer à minha família, em especial aos meus pais, porque sem eles eu nunca teria chegado até aqui, e à minha irmã, pelo incentivo ímpar e por, ao longo da vida, servir de exemplo a seguir.

Este documento não permite um agradecimento devido a todos aqueles que o merecem, mas a todos, um sincero obrigado!

Filipe Azevedo

"It is a mistake to think that moving fast
is the same as actually going somewhere."

Steve Goodier

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrónimos e Abreviaturas

API	*Application Programming Interface* (Interface de Programação de Aplicações)
CPU	*Central Processing Unit* (Unidade Central de Processamento)
FEUP	Faculdade de Engenharia da Universidade do Porto
GNSS	*Global Navigation Sattelite System* (Sistema de Navegação Global por Satélite)
GPU	*Graphics Processing Unit* (Unidade de Processamento Gráfico)
hLBP	*Histogram of Local Binary Pattern* (Histograma de Padrões Binários Locais)
HPC	*High Performance Computing* (Computação de Alta *Performance*)
HSV	*Hue, Saturation and Value* (Tonalidade, Saturação e Valor)
IDE	*Integrated Development Environment* (Ambiente de Desenvolvimento Integrado)
INESC TEC	Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência
LBP	*Local Binary Pattern* (Padrões Binários Locais)
libSVM	*Library for Support Vector Machines* (Biblioteca de Máquina de Vetores de Suporte)
LRF	*Laser Range Finder* (Localizador de Alcance a Laser)
NPU	*Natural Processing Unit* (Unidade de Processamento Natural)
OpenCV	*Open Source Computer Vision Library* (Biblioteca *Open Source* de Visão Computacional)
RBF	*Radial Basis Function* (Função de Base Radial)
RGB	*Red, Green and Blue Color Space* (Espaço de Cores Vermelho, Verde e Azul)
ROMOVI	*Modular and Cooperative Robot for Vineyards* (*Robot* Modular e Cooperativo para Vinhas)
ROS	*Robot Operating System* (Sistema Operativo para *Robots*)
SM	*Streaming Multiprocessor* (não traduzido)
SP	*Streaming Processor* (não traduzido)
SV	*Support Vector* (Vetor de Suporte)
SVM	*Support Vector Machine* (Máquina de Vetores de Suporte)
UTAD	Universidade de Trás-os-Montes e Alto Douro
VineSLAM	*Vine Simultaneous Localization And Mapping* (Localização e Mapeamento Simultâneos em Contexto de Vinha)
ViTruDe	*Vine Trunks and Masts Detector* (Detetor de Troncos e Mastros em Vinhas)

Capítulo 1

Introdução

Neste primeiro capítulo introdutório, é feita uma apresentação do tema da presente dissertação: "Paralelização de Algoritmos de Processamento de Imagem em Contexto de Vinha". É referida a contextualização da dissertação na secção 1.1, bem como os objetivos principais que pretendem ser alcançados com a sua realização na secção 1.2. De seguida, na secção 1.3 são também mencionadas as motivações que levaram à proposição deste tema. Para finalizar este capítulo, na secção 1.4, é feita uma estruturação da dissertação, mencionando os pontos sobre os quais esta se vai focar nos restantes capítulos.

1.1 Contexto

A dissertação "Paralelização de Algoritmos de Processamento de Imagem" tem por base a análise de algoritmos de visão computacional e a sua otimização e implementação em GPU, de modo a permitir reduzir os tempos de processamento por *frame* e por conseguinte aumentar a eficiência dos processos de localização de uma plataforma robótica.

A automatização de processos tem cada vez mais procura nas mais variadas áreas industriais, tendo também a área da agricultura a necessidade de automatizar processos. Desta necessidade agrícola, surgiu o algoritmo ViTruDe, um algoritmo de processamento de imagem para deteção de elementos naturais nas videiras, nomeadamente os mastros e troncos. Este algoritmo foi desenvolvido no âmbito do projeto *Modular and Cooperative Robot for Vineyards* (ROMOVI), que tem como objetivo o desenvolvimento de componentes robóticas, como uma plataforma móvel extensível - tipicamente, um *robot*. Esta plataforma permitirá no futuro criar soluções comerciais capazes de executar operações de monitorização e logística de forma autónoma em vinhas de encostas, respondendo a problemas nesta área. O ROMOVI pretende desenvolver esta plataforma de forma modular e *low-cost*, de modo a automatizar os processos logísticos e de monitorização em contexto de vinhas de montanha [9].

A plataforma mencionada acima é o *robot* AgRob V16, na qual estarão implementados algoritmos de ajuda à sua localização como o já mencionado ViTruDe.

1.2 Objetivos

Esta dissertação tem como principal objetivo a otimização de código em GPU, através de um método de paralelização de algoritmos de processamento de imagem. Este método consiste na partição de um problema computacional em segmentos mais pequenos e simples, que possam correr concorrentemente. Isto permitirá melhorar a velocidade da computação. Este método será abordado com maior rigor na revisão bibliográfica, no capítulo 2.

Os algoritmos de processamento de imagem são essenciais para um sistema de localização robusto, e a necessidade de melhoria da *performance* computacional surgiu como forma de tornar o AgRob numa plataforma robótica capaz de ter um impacto real na área da agricultura de vinhas de encosta, sendo economicamente viável e ao mesmo tempo com um desempenho satisfatório, por isso, a única forma do AgRob poder cumprir estes requisitos é fazendo as suas operações nas vinhas de uma forma mais rápida, utilizando para isso a capacidade de paralelização dos GPUs.

Existem diferentes soluções já testadas para monitorização de vinhas planas, porém, neste caso, com vinhas inclinadas, o processo de localização de uma plataforma robótica torna-se bastante mais complexo porque a sensorização, perceção e interpretação das colheitas têm de ser eficientes, precisas e robustas devido ao ambiente incerto e não estruturado [10].

1.3 Motivação

Sabendo que existe uma constante evolução e inovação da automatização e que esta vai tendo um papel cada vez mais determinante por toda a indústria, o setor agrícola não é uma exceção. Havendo já avanços relativamente à monitorização e colheita autónoma por *robots* térreos em terrenos agrícolas planos (apesar de ainda haver poucos *robots* com aplicações agrícolas no mercado [10]), surgiu a necessidade de implementar um processo similar, mas em terrenos de vinhas de encosta, tendo em conta fatores competitivos como o custo e qualidade.

A agenda de pesquisa estratégica europeia para robótica [11] afirma que os *robots* conseguem melhorar a eficiência e competitividade da agricultura. Portugal é um país de enorme tradição e potencial nesse ramo, nomeadamente na região do Douro, com as famosas videiras das quais o tão aclamado Vinho do Porto é produzido. Esta produção constitui o fator económico de exportação mais importante da região e dos mais importantes do país [12], por isso, é natural a aposta em meios de automatizar processos de modo a tornar este ramo ainda mais rentável. É aqui que entra o papel do ROMOVI.

Tomando novamente como exemplo as vinhas das margens do rio Douro, variadíssimos fatores podem influenciar uma colheita, tornando-a rentável ou, por outro lado, economicamente desastrosa. Um destes fatores a ter em conta é a condição climatérica da região [13], que, sendo desfavorável, pode fazer com que as plantas ganhem cores pouco habituais que possam representar sinais de doença. Neste caso, a plataforma robótica AgRob será de grande utilidade visto que poderá detetar doenças das plantas, a necessidade de água por parte da planta e verificar o estado da vegetação (como detetar a presença de pragas, por exemplo).

Face ao exposto, a principal motivação desta dissertação é ajudar no desenvolvimento da vinicultura, na medida em que se pode, através de meios robóticos autónomos cujo planeamento futuro é a implementação em *robots* comerciais, recolher informação valiosa que poderá oferecer uma ajuda considerável ao setor agrícola.

1.4 Estrutura da Dissertação

Para além do presente capítulo de introdução ao tema, esta dissertação contém mais cinco capítulos, descritos a seguir:

- Capítulo 2: É descrita a revisão bibliográfica da dissertação, onde é mencionada a informação corrente e atual sobre o tema, bem como a inclusão de resultados e contribuições relevantes na área;
- Capítulo 3: É explicada a abordagem e implementação das estratégias e metodologias aplicadas durante a realização dos trabalhos da dissertação;
- Capítulo 4: São exibidos os testes efetuados ao algoritmo implementado, bem como os resultados mais relevantes obtidos durante a sua implementação;
- Capítulo 5: É apresentado um resumo das tarefas exercidas durante a realização da tese e são expostas as principais conclusões obtidas, sendo também exibidas algumas considerações acerca das mesmas.

Capítulo 2

Revisão Bibliográfica

Este capítulo será dedicado à revisão bibliográfica desta dissertação, e foi realizado com base na consulta e pesquisa nas mais diversas plataformas disponíveis. O principal objetivo deste capítulo é o estudo de toda a informação relevante ao tema da dissertação. Após a pesquisa estar concluída, foi possível destacar quatro secções principais, respetivamente: "Robótica na Agricultura", 2.1; "Algoritmo ViTruDe", 2.2; "Computação Paralela", 2.3; "Ferramentas de Programação", 2.4. Estas áreas serão abordadas nos subcapítulos seguintes.

Na primeira secção são abordados os trabalhos de robótica da área de processamento de imagem que foram feitos na área de agricultura, de modo a estudar a sua possível integração no tema desta dissertação.

A segunda secção foca-se no estudo da abordagem de processamento visual desta dissertação e todos os seus aspetos envolventes, nomeadamente a plataforma robótica utilizada, no tópico 2.2.1, bem como o algoritmo ViTruDe, com a análise de testes e seus respetivos resultados, no tópico 2.2.2.

Na terceira secção são abordados os tópicos referentes à arquitetura de GPUs e a sua respetiva computação paralela, de modo a perceber melhor o porquê de cada vez mais se utilizar estas unidades de processamento para realizar as mais variadas tarefas de computação. O tópico 2.3.1 refere os principais detalhes das diferentes arquiteturas computacionais, enquanto que por sua vez, o tópico 2.3.2 analisa em maior detalhe a computação heterogénea e compara objetivamente o desempenho de diferentes algoritmos de processamento de imagem em GPU por oposição ao CPU, explicando a sua evolução.

A quarta e última secção foca-se nas ferramentas de programação abordadas na dissertação, sendo analisados os motivos e vantagens de se programar paralelamente fazendo uso destas ferramentas.

2.1 Robótica na Agricultura

Com a contínua procura de comida para a crescente população, para além da redução de terrenos férteis de cultivo, mudanças climáticas e instabilidade política, a indústria da agricultura continua a procurar novas formas de melhorar a produtividade e sustentabilidade [14].

Uma das estratégias para otimizar a produção agrícola consiste em recorrer a *robots* autónomos para auxílio nessas atividades, e o projeto desta dissertação pertence, certamente, a este ramo. Apesar da quantidade de plataformas robóticas com aplicações agrícolas no mercado estar ainda longe dos níveis desejados [10], já houve, efetivamente, vários avanços e trabalhos nesta área.

Considerando que esta dissertação tem por base o processamento de imagem que permita dar a sua contribuição ao setor agrícola, esta secção vai estudar algum do trabalho realizado nessa vertente, ou seja, trabalho de processamento de imagem em robótica com aplicações agrícolas. Este processo permitirá à agricultura tornar-se mais precisa, e assim ajudar os agricultores a tomar decisões acerca da seleção de sementes, na produção de diferentes culturas, na monitorização de doenças, no controlo de ervas daninhas e no uso de pesticidas e fertilizantes de forma mais eficaz [15]. Na Tabela 2.1 são referidas algumas técnicas e etapas de processamento visual que podem ter aplicação neste ramo.

Tabela 2.1: Técnicas de processamento de imagem com aplicação na agricultura [7]

Técnica	**Significado**
Aquisição de imagem	Processo de obtenção de uma imagem digital a partir de uma fonte física, utilizando sensores.
Conversão a escala cinza	Processo de conversão de uma imagem digital a cores para uma imagem que possui *pixels* de um valor único de intensidade.
Extração do fundo	Separação do fundo da imagem, obtendo objetos presentes no primeiro plano.
Melhoria de imagem	Melhoria da perceção dos detalhes da imagem para futura análise humana.
Análise de histograma	Análise dos *pixels* da imagem em termos de valores de intensidade, como picos ou vales formados pela relação de frequência e intensidade dos mesmos.
Segmentação binária	Separação dos objetos de primeiro plano de uma imagem dos de fundo, numa imagem a preto e branco.
Segmentação de cor	Separação dos objetos de uma imagem por cor, consoante regiões de interesse.
Filtragem de imagem	Processo de distorção de uma imagem de uma forma pretendida, usando filtros.
Extração de características	Processo de transformação de diferentes tipos de dados num sistema de coordenadas.
Inscrição de imagem	Processo de definição de um conjunto de características que representam informação importante para análise e classificação.
Deteção de objetos	Processo de procura de objetos reais como ervas, plantas ou insetos em imagens ou sequências de vídeo.
Análise de objetos	Processo de extração de informação útil a partir de imagens.

Uma aplicação computacional de visão utiliza técnicas de processamento que envolvem cinco passos básicos: aquisição da imagem, pré-processamento, segmentação, deteção de objetos e classificação. Existem vários exemplos de estudos sobre o processamento de imagens e suas aplicações práticas no contexto agrícola [7], como por exemplo o estudo da aplicabilidade de lógica Fuzzy para deteção e reconhecimento de ervas daninhas, cujo resultado foi uma redução de 15 a 64% no uso de herbicidas [16].

A seguir são apresentadas informações adicionais acerca das técnicas de processamento de imagem descritas na Tabela 2.1, bem como referenciadas aplicações e trabalhos ligados à área agrícola realizados com ajuda e suporte dessas mesmas técnicas.

Conversão a escala cinza:

Depois de adquirida, uma imagem tem de ser pré-processada através da conversão a escala cinza. Esta conversão foi considerada como um passo intermédio em modelos de avaliação de comida [17, 18], nomeadamente em aplicações para comida como fruta, peixe, carne, vegetais, sementes, entre outras. Este passo é relevante porque a maioria dos descritores de imagens foram desenhados para operar em escalas de cinzas. Extração em imagens RGB (espaço de cores *Red*, *Green* e *Blue*) não tem provado ser uma mais valia e a relação entre custo de processamento e robustez não é relevante.

Extração do fundo:

Em aplicações onde o fundo tem menor importância relativamente aos elementos de primeiro plano, é preferível extrair esse mesmo fundo dessas imagens. Possuir objetos em fundos completamente diferentes é um modo de os extrair facilmente. O resultado desta operação é um nível de distribuição cinzento não uniforme entre objetos de interesse e o fundo da imagem [17, 18].

Melhoria de imagem:

Técnica aplicada quando uma imagem se encontra bastante danificada com ruído ou pobre em contraste [19]. Existem várias operações que fazem a melhoria de uma imagem, como operações morfológicas, filtros e operações *pixel* a *pixel*, que são usadas para minimizar irregularidades na imagem causadas por uma iluminação inadequada.

Análise de histograma:

O uso de um histograma pode refletir o efeito direto causado por iluminação onde o contraste é uma característica de identificação da cor verde na imagem [20], por exemplo.

Segmentação binária:

Através desta técnica, é possível detetar vários elementos de interesse em imagens agrícolas, como é o caso de um estudo de deteção de defeitos em superfícies de maçãs [1], demonstrado na Figura 2.1. Este estudo teve um rendimento de 96% a detetar cortes e outros danos nas várias combinações de imagens utilizadas. Contudo, a presença de sombras e iluminação imprevisível pode comprometer a fiabilidade deste método.

Segmentação de cor:

Tal como a segmentação binária, a segmentação de cor é uma técnica bastante útil que permite detetar regiões de interesse em imagens com base nas suas cores. Já foi utilizada, por exemplo,

para separação de fruta baseada na sua cor, embora com pouca robustez devido à incapacidade de separar ruído branco na imagem [21].

Figura 2.1: Segmentação binária em maçãs, para deteção de defeitos [1].

Filtragem de imagem:

Esta técnica é muito versátil devido à quantidade de filtros diferentes que podem ser utilizados para obter resultados diferentes. Com base nesta técnica, um algoritmo foi utilizado para separar culturas de cevada, milho e cereais, incluindo ervas e terra (na Figura 2.2 estão exemplos de separação de duas classes de plantas à esquerda, e duas diferentes classes de plantas, com terra, à direita) [2].

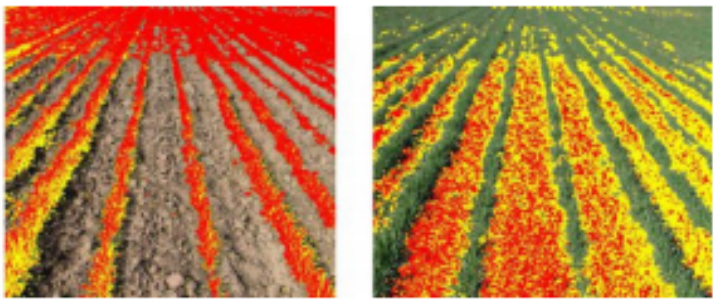

Figura 2.2: Resultados de uma filtragem de imagem para separar duas diferentes classes de plantas [2].

Extração de características:

Com esta técnica foi possível fazer uma classificação de ervas baseada nas suas características morfológicas. Esta classificação teve resultados melhores do que triagens manuais [22].

Inscrição de imagem:

Um exemplo de uma aplicação desta técnica na agricultura é o uso de sensores espectrais à distância para deteção de doenças em tomates. Este sistema utilizou um espectrómetro com sensores infravermelhos para capturar as imagens dos tomates. Com isto foi possível detetar ativamente doenças e ajudar na sua classificação e identificação [23].

Deteção de objetos:

Esta técnica foi aplicada num sistema que foi desenvolvido para deteção e extração de terra, plantações e ervas em imagens. O sistema obteve percentagens de correlação de respetivamente

84%, 96% e 84% para biomassa, imagens de cereais de amadurecimento tardio e imagens de milho [24].

Análise de objetos:

A deteção da forma e cor de maçãs maduras em imagens de árvores foi fruto de um estudo recentemente, baseado em análise de convexidade a três dimensões [25]. Esta aplicação teve 94% de fiabilidade em maçãs cujas bordas foram identificadas com o filtro de Canny[1].

Relativamente ao processamento de imagem para fins de localização de plataformas robóticas, um método preliminar baseado em imagem tinha por base um procedimento que detetava as filas das culturas plantadas usando uma segmentação binária em imagens próximas do infravermelho[2] de filas de culturas [26]. A segmentação binária torna-se pouco fiável quando existem sombras parciais ou ervas consideráveis entre as linhas de colheitas. A maioria dos métodos de visão monocular primeiro executam a segmentação binária entre as plantas e o solo (normalmente baseada em cor) e depois utilizam métodos como a transformada de Hough ou regressão linear. Estes métodos tornam-se por vezes computacionalmente pesados para serem executados em tempo real, para além de que não são totalmente robustos em ambientes incertos, podendo levar a uma incorreta extração de linhas e consequente falha na navegação. A robustez destes métodos pode ser melhorada filtrando de forma conjunta a odometria das rodas do *robot* e a captação de imagens das filas de colheitas da sua câmara usando um filtro de partículas [27].

Uma abordagem que evita a segmentação é a procura em pequenas porções da imagem em escala cinzenta [28]. Desta forma o campo de visão é reduzido a pequenos "pedaços" ou listas horizontais, não estimando, ou tendo em conta, variações na posição da câmara, de modo a evitar "manchas", o que aconteceria se fossem abordados campos de visão superiores.

Na Figura 2.3 pode observar-se um esquema simplificado de um algoritmo de localização robótica em ambientes de agricultura para deteção de filas de colheitas [3]. Aqui, estas filas são modelizadas como texturas paralelas planas, o que permite retirar características específicas como cor, espaçamento e periodicidade. É usado um método de extração da direção da textura paralela dominante, a qual é usada para corrigir o *offset* lateral do *robot*. O comportamento da câmara também é estimado, sendo o *robot* operado num sistema de coordenadas local, alinhados com as filas e centrado na sua posição quando o algoritmo começa.

Concluindo, existem técnicas de processamento de imagem que são utilizadas em vastas áreas do ramo agrícola, tem é que se ter em atenção o facto de que se necessita de escolher as técnicas mais adequadas para diferentes casos. Nesta dissertação, o pretendido é a deteção de características nas vinhas de encosta, algo bastante diferente dos casos até aqui descritos, pois as vinhas de encosta têm características únicas e oferecem dificuldades acrescidas relativamente à agricultura

[1] Filtro de Canny é um conhecido operador para deteção de bordas e arestas em processamento de imagens.

[2] Imagens próximas do infravermelho, do inglês *near-infrared*, são imagens próximas do espectro visível que pertencem ao espectro de infravermelho.

plana. O estudo deste tipo de algoritmo para deteção de características naturais para efeitos de localização será abordado com detalhe no capítulo 2.2.

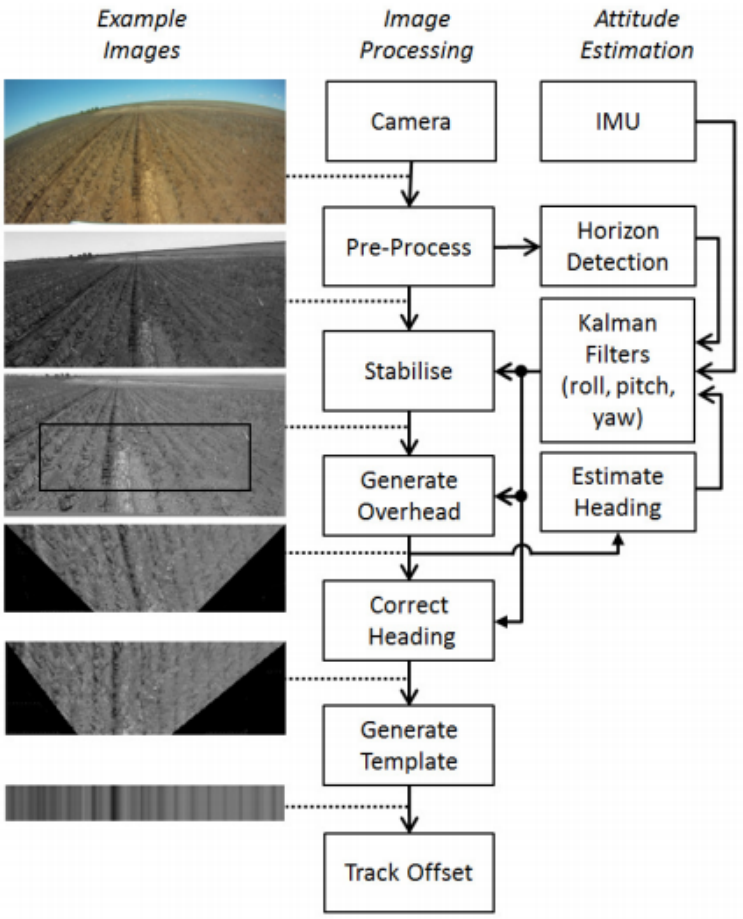

Figura 2.3: Algoritmo de localização de uma plataforma robótica com base em processamento de imagem [3].

2.2 Algoritmo ViTruDe

Desenvolver *robots* terrestres de monitorização de vinhas é uma tarefa bastante complexa [5] e quando se trata de vinhas inclinadas a dificuldade e complexidade sobe exponencialmente. Estas regiões inclinadas diferem das planas na medida em que:

- Características do terreno bloqueiam os sinais e fazem multi-reflexão, o que faz com que os sistemas GNSS fiquem suscetíveis a falhas e nem sempre disponíveis;
- Condições do terreno inviabilizam os *dead reckoning sensors*[3];
- Inclinação do terreno faz com que o planeamento da trajetória do *robot* dependa de mapas precisos e de uma boa observação da sua postura.

[3]Sensores que calculam a posição usando a posição anteriormente determinada.

Com base nestes terrenos, e tendo em mente as vinhas típicas das margens do rio Douro (ver Figura 2.4[4]), o algoritmo a utilizar para localização não pode recorrer ao sistema de localização global GNSS, pois, para além da irregularidade e altitude das vinhas, que limita o número de satélites que podem visualizar a posição [29], estas também dificultam a exatidão e rigor com que o GNSS estima a localização do *robot*. Para solucionar este problema, e atendendo à situação específica, seria necessário um sistema de localização que fosse independente do GNSS e também que estivesse disponível a tempo inteiro, fornecesse informações em tempo real, fosse seguro e eficiente.

Figura 2.4: Vinhas inclinadas típicas das margens do rio Douro.

O *Vine Simultaneous Localization And Mapping* (VineSLAM) [29, 30] é um sistema de localização para vinhas inclinadas robusto a falhas do GNSS que usa um mapa híbrido para aumentar a precisão e robustez da localização. A localização topológica do *robot* sofre atualização considerando pontos de referência artificiais de baixo custo [31]. Em localização métrica, o VineSLAM considera as características naturais [29] das vinhas de modo a obter melhor fiabilidade. Estas características naturais são os troncos e mastros das videiras, que existem em grande quantidade nas vinhas. Todavia, um sistema que consiga detetar corretamente estes elementos necessita de vários sensores, como *Laser Range Finders* (LRFs) ou câmaras. O uso singular de sensores LRF não é aconselhado porque pode confundir regiões de terra e erva com troncos e mastros de videiras [30]. Um sistema de processamento de imagem consegue fornecer informação rica e complementar por forma a atingir uma deteção robusta. Aqui entra o ViTruDe, algoritmo responsável pela deteção de elementos naturais nas videiras, nomeadamente os mastros e troncos.

2.2.1 Plataforma Robótica

O Centro de Robótica Industrial e Sistemas Inteligentes, do Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência (INESC TEC), desde 2014 que delineou uma linha de investigação denominada de AgRob [4], que se foca no desenvolvimento de soluções de robótica

[4]Figura retirada de: www.dourovalley.eu

Figura 2.5: Plataforma robótica AgRob V16 [4].

para aplicação à agricultura, mais propriamente, ao contexto de vinhas inclinadas, tal como foi abordado acima.

A plataforma robótica em que está aplicado o algoritmo ViTruDe no qual se foca a presente dissertação é o AgRob V16 (existem também outras versões anteriores de plataformas AgRob). Esta plataforma robótica (Figura 2.5) está a ser desenvolvida de forma modular para a execução de tarefas relacionadas com esta área de agricultura, como monitorização, pulverização, poda e colheita. Estas tarefas estão ilustradas na Figura 2.6.

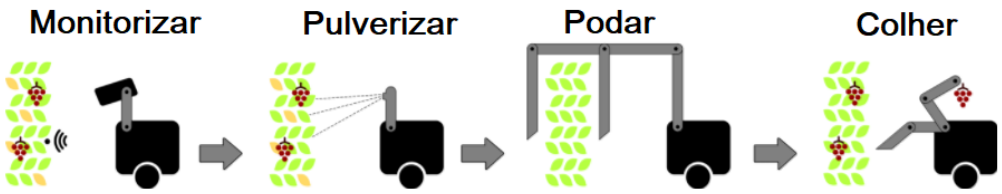

Figura 2.6: Tarefas que o AgRob V16 é capaz de realizar. Adaptado de [4].

Este *robot* de pequeno/médio porte foi então totalmente desenhado para operar de forma completamente autónoma, robusta e em segurança, no contexto de vinhas de encosta.

Uma das unidades de processamento utilizadas nesta plataforma é o NVIDIA Tegra K1[5] com as especificações encontradas na Tabela 2.2.

[5]http://www.nvidia.com/object/tegra-k1-processor.html

Tabela 2.2: Especificações do processador NVIDIA Tegra K1

	TEGRA K1
GPU *NVIDIA® Kepler™ Architecture*	 192 NVIDIA CUDA® Cores
CPU Núcleos e Arquitetura CPU Velocidade Máxima do *Clock*	 NVIDIA 4-Plus-1™ Quad-Core ARM Cortex-A15 "r3" 2.3 GHz
Memória Tipo de Memória Tamanho máximo	 DDR3L e LPDDR3 8 GB (com extensão de endereço de 40 bits)
Display LCD HDMI	 3840x2160 4K (UltraHD, 4096x2160)
Package Tamanho/Tipo Processo	 23x23 FCBGA 16x16 S-FCCSP 15x15 FC PoP 28 nm

2.2.2 Algoritmo ViTruDe - Abordagem em CPU

ViTruDe é a abordagem de processamento de imagem realizada para detetar os elementos naturais das vinhas [5], passo fundamental para complementar os mapas usados no VineSLAM.

O ViTruDe, em CPU, tem três passos principais (Figura 2.7):

- *Keypoints Extractor*;
- *KeyPoints Search and Region Descriptor Extractor*;
- Classificação por parte da *Support Vector Machine* (SVM).

Dada uma imagem, primeiramente são detetados os pontos-chave utilizando o operador Sobel, onde uma pequena região vai ser analisada para detetar a presença de troncos. Isto reduz o custo computacional na análise de todas as imagens.

É depois extraído um descritor na região circundante de cada ponto-chave. Este descritor é a entrada do classificador SVM. Tendo como base o treino efetuado, o SVM irá classificar se se está na presença de algum tronco ou mastro nessa pequena janela da imagem. A região tem um tamanho fixo de 30x100 *pixels*, escolhido para capturar um tronco comum a uma distância de um ou dois metros com uma câmara de 640x480 *pixels* e preencher 60 a 90% da região da imagem.

O descritor é baseado no operador *Local Binary Pattern* (LBP[6]). O LBP original trabalha tipicamente com uma grelha de *pixels* 3x3 para um dado *pixel* numa imagem de escala-cinza,

[6]LBP (*Local Binary Pattern*) é um poderoso operador classificador de texturas com base na escala-cinza.

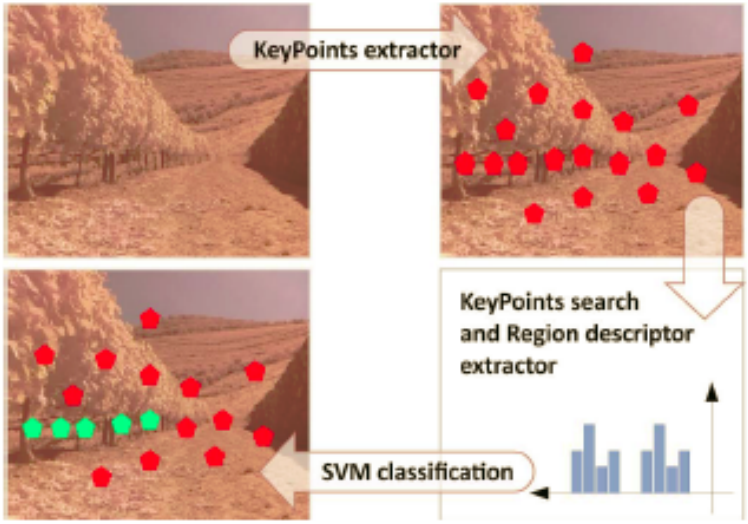

Figura 2.7: Passos principais do algoritmo ViTruDe [5].

em que é comparado o valor do cinzento do *pixel* central e dos seus vizinhos, dentro da grelha 3x3, podendo ser utilizados diferentes tamanhos de grelha. É calculado um código LBP binário através da implementação de um limiar à grelha de *pixels*. Este limiar é o valor do *pixel* central, que determina a atribuição de um valor binário aos *pixels* em questão, obtendo assim um código LBP. É de notar que os *pixels* que não estão ao alcance da grelha 3x3 têm o seu valor de cinzento estimado por interpolação. É então criado um histograma a partir de todos os padrões binários de cada *pixel* para descrever a textura da imagem (chamado *Histogram of Local Binary Pattern* - hLBP) - Equações (2.1) e (2.2).

$$H(k) = \sum_{m=1}^{M} \sum_{n=1}^{N} f(LBP_{P,R}(m,n),k), k \in [0,K], \text{ onde K é o valor máximo do LBP.} \tag{2.1}$$

$$f(x,y) = \begin{cases} 1, & x = y \\ 0, & \text{outro} \end{cases} \tag{2.2}$$

Baseado no hLBP, dois tipos de descritores são desenvolvidos e construídos: *hLBP_by_colour* e *hLBP_plus_colour*, Figura 2.8.

O *hLBP_by_colour* tem um histograma por espaço de cor. Este espaço de cor é dividido em *n* cores. O tamanho do descritor é dado pelo número de códigos LBP vezes o número de espaços de cor. Para reduzir o tamanho, a variante uniforme do LBP tem 8 espaços de cor. Neste descritor, para cada *pixel* é detetado o espaço de cor relacionado e a barra do histograma do código do LBP é incrementada para esse *pixel*.

Contrariamente, o *hLBP_plus_colour* tem dois histogramas, uma para os códigos LBP e outro para espaços de cor. O espaço de cores é também dividido em *n* cores e o tamanho do descritor é dado pela soma entre o número de códigos LBP e o número de espaços de cor. Para cada

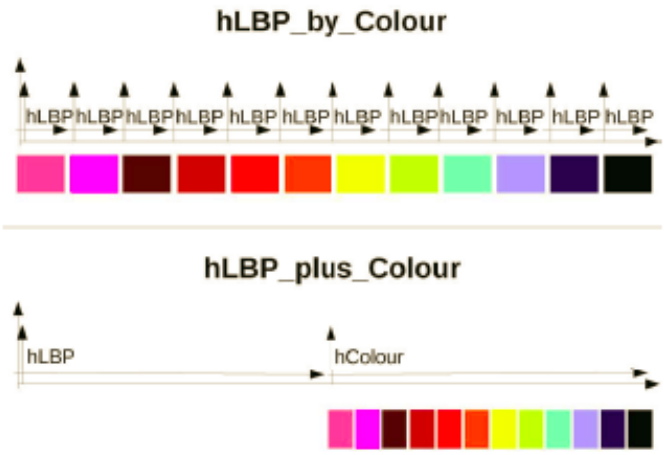

Figura 2.8: Dois tipos de descritores utilizados no ViTruDe [5].

pixel é detetado o espaço de cor relacionado, de modo a incrementar essa barra do histograma, e posteriormente extraído o código LBP para incrementar a barra do histograma relacionada com esse código.

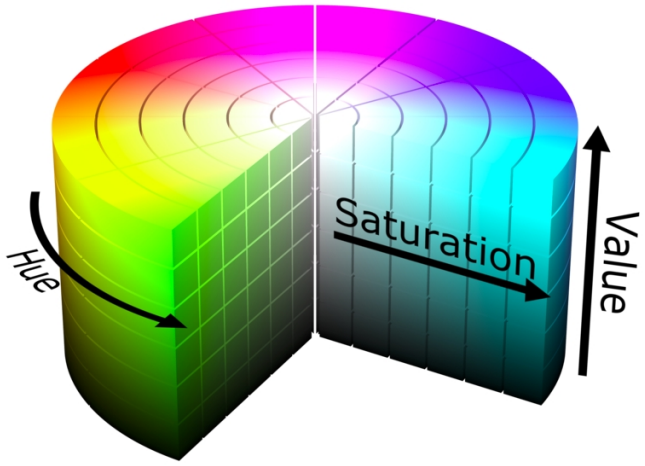

Figura 2.9: Modelo de cores HSV mapeado num cilindro.

Os espaços de cor representam a dimensão "Hue" de uma imagem HSV[7] e são extraídos a partir desta. A Figura 2.9[8] permite perceber melhor a estrutura tridimensional destas imagens. Este tipo de imagens é amplamente utilizado para fins científicos e permite fazer a perceção de como a visão humana faz a criação de cores, modelando a forma como cores diferentes se misturam [32, 33]. Neste modelo, a dimensão das tonalidades ("Hue"), ou seja, das gamas de cor, são organizadas

[7] Abreviatura para a decomposição de uma imagem em três dimensões: Hue, Saturation e Value. Em português: Tonalidade, Saturação e Valor.

[8] Figura retirada de: http://isometricland.net/home.php

de forma radial, em torno de um eixo central de cores neutras (preto no fundo e branco no topo, como mostra a Figura 2.9).

2.2.2.1 Testes e Resultados

Os treinos, testes e validações foram feitos com dados reais, obtidos pela plataforma robótica AgRob [29], em três dias diferentes e com céu limpo, ao nascer-do-Sol e ao pôr-do-Sol.

Para o treino e teste do SVM, foram criados seis conjuntos de dados, construídos a partir de *frames* de imagens de três configurações distintas de filtros da câmara: NoIR[9] sem filtro, NoIR com filtro e câmara RGB. A resolução da câmara é de 640x480 *pixels*, com um *framerate* de 5 *frames* por segundo. Para cada configuração de filtro, foram obtidos 200 *frames* de imagens. Estas imagens foram divididas em imagens mais pequenas de 30x100 *pixels* de modo a se obter 3000 para cada classe.

Para validar a *performance* do SVM em múltiplas classes, foram criados seis conjuntos de dados, como havia sido dito acima, divididos em dois grupos:

- Três conjuntos de dados construídos a partir de duas classes - Troncos e Outros;
- Três conjuntos de dados construídos a partir de quatro classes - Troncos, Céu, Chão e Outros.

A Tabela 2.3 mostra um sumário de cada conjunto de dados construídos.

Tabela 2.3: Conjunto de dados criado para treinos e testes do SVM

Filtro	**Classes**	**Nº imagens por classe**	**Treino/Teste**
$NoIRF_2$	Troncos e Outros	3000	2500/500
$NoIRI_2$	Troncos e Outros	3000	2500/500
RGB_2	Troncos e Outros	3000	2500/500
$NoIRF_4$	Troncos, Céu, Chão e Outros	3000	2500/500
$NoIRI_4$	Troncos, Céu, Chão e Outros	3000	2500/500
RGB_4	Troncos, Céu, Chão e Outros	3000	2500/500

Vários testes foram realizados com estes seis conjuntos de dados e os dois tipos de descritores [5]. Estes testes foram realizados num computador com um processador Intel® Core™ i7-4700HQ, 12 GB de RAM e sistema operativo Ubuntu 14.04 LTS, com testes em SVM de núcleos lineares, polinomiais e *Radial Basis Function* (RBF).

A Figura 2.10 mostra três tipos de imagens de entrada, enquanto que a Figura 2.11 mostra os resultados obtidos em CPU com o descritor *hLBP_plus_colour*, SVM linear de quatro classes. Os retângulos vermelhos assinalam o céu, os verdes assinalam os troncos/mastros e os azuis assinalam o chão.

Figura 2.10: Imagens usadas para os testes de processamento de vídeo [5].

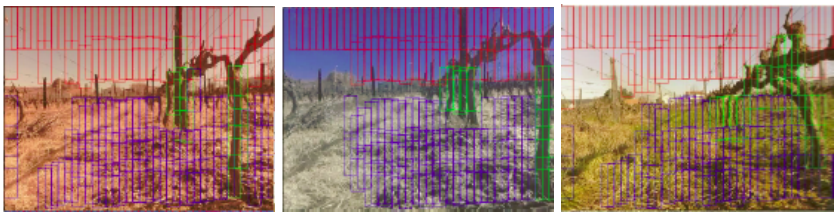

Figura 2.11: Resultados obtidos com o *hLBP plus colour* utilizando quatro classes [5].

Concluindo, o algoritmo ViTruDe foi construído a pensar em tarefas de vinhas de montanha, tendo em conta as suas dificuldades, e foi executado por uma plataforma robótica que tenta ao máximo otimizar a relação custo-eficiência de todo o processo. O que se verificou com os testes realizados foi que todas as configurações testadas tiveram uma grande eficácia a nível de perceção visual. A nível de tempo de processamento, uma análise completa a toda a *frame* e não apenas aos pontos-chave detetados, no ambiente de trabalho descrito acima, demora cerca de 10 minutos a executar por *frame*, o que torna o processo demasiado lento para ter aplicação nas tarefas do AgRob, que se pretende que sejam executadas o mais próximo possível de tempo real.

Como forma de otimização do processo e devido à necessidade de uma solução que seja completamente em tempo real, serão estudadas no subcapítulo 2.3 arquiteturas computacionais alternativas ao CPU que consigam um processamento mais rápido e eficaz.

2.3 Computação Paralela

O ramo de computação de alta *performance* (*High Performance Computing* - HPC) está sempre em constante mudança, com novas tecnologias a serem desenvolvidas a cada momento, e por isso este ramo desenvolve-se de acordo com esses avanços. Fruto deste avanço tecnológico e das crescentes necessidades de processamento, resultam os GPUs, unidades de processamento gráfico capazes de processar dados paralelamente.

A computação paralela pode ser definida como o uso simultâneo de múltiplos recursos computacionais (núcleos ou computadores), para resolver problemas concorrentemente.

A seguir será apresentada a evolução dos GPUs, que tradicionalmente eram apenas usados como coprocessadores gráficos, mas hoje em dia são usados para as mais variadas funções. O rápido desenvolvimento de *hardware* gráfico levou a que os GPUs e a consequente programação

[9] Sem infravermelhos.

paralela passassem a ter um uso extensivo tanto nos ramos da ciência e engenharia, como em aplicações comerciais [34].

2.3.1 Arquiteturas Computacionais

Um dos maiores objetivos da computação, desde a sua origem, foi, e continua a ser, a velocidade. Qualquer programa, para ser eficiente, tem de ser executado rapidamente. Foi com base nesta busca por maior velocidade que os GPUs começaram a desempenhar funções até aí apenas designadas para os CPUs, os componentes chaves do HPC, usualmente denominados de núcleos.

Hoje em dia, a maioria dos computadores utiliza a arquitetura "Harvard", desenvolvida na Universidade de Harvard entre 1937 e 1944 e visualizada na Figura 2.12, tendo como principal característica o acesso à memória de dados de modo separado em relação à memória de programa. Isto permite que um processador possa aceder às duas memórias simultaneamente, resultando assim o conceito de paralelismo [35].

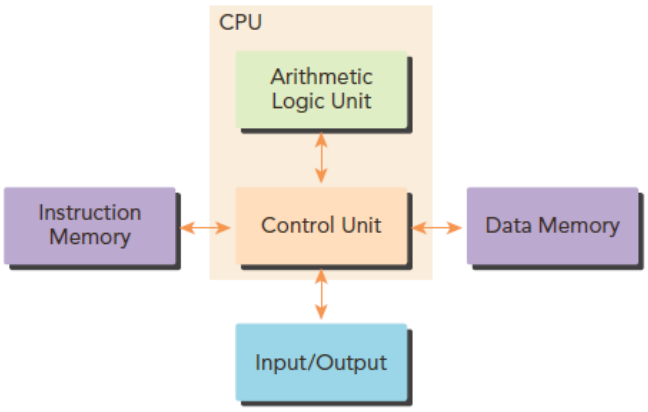

Figura 2.12: Arquitetura Harvard [6].

Antigamente, nos dias embrionários dos computadores, apenas havia um núcleo por computador, numa arquitetura chamada de *uniprocessor*. Hoje em dia, a tendência no *design* de *chips* de computadores é integrar vários núcleos num só processador, termo chamado de *multicore*. Isto permite o suporte de programação paralela ao nível da arquitetura, sendo que, deste modo, a programação pode ser vista como o processo de mapeamento da computação de um problema pelos diferentes núcleos disponíveis, de modo a que a programação paralela seja possível de obter [6].

As arquiteturas dos sistemas de computação sempre foram, e sempre serão, bastante influenciadas pelas tendências e capacidades das tecnologias de *hardware* e *software*. Há várias formas de classificar as arquiteturas dos computadores.

Um esquema de classificação utilizado de forma abrangente é o *Flynn's Taxonomy*, que divide as arquiteturas em quatro diferentes tipos de acordo com a maneira como as instruções e a informação flui pelos núcleos (Figura 2.13), dos quais se destacam:

- *Single Instruction Single Data* (SISD);
- *Single Instruction Multiple Data* (SIMD);
- *Multiple Instruction Single Data* (MISD);
- *Multiple Instruction Multiple Data* (MIMD).

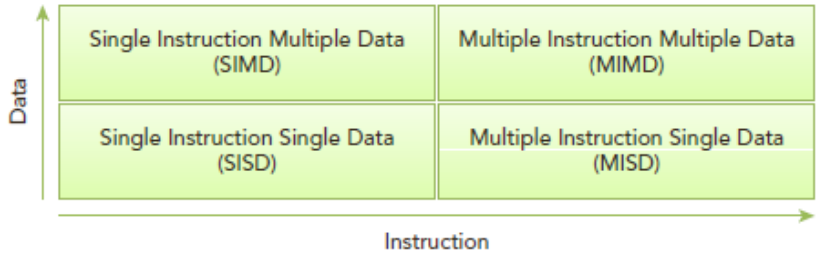

Figura 2.13: *Flynn's Taxonomy* [6].

Single Instruction Single Data refere-se ao computador tradicional e original, com apenas um núcleo. É apenas executado um fluxo de instruções em qualquer momento e as operações são realizadas num só fluxo de dados.

Single Instruction Multiple Data é um tipo de arquitetura paralela, havendo vários núcleos no computador. Todos os núcleos executam o mesmo fluxo de instruções, cada um operando em diferentes fluxos de dados. A maior vantagem desta arquitetura é que ao escrever código em CPU, é possível pensar-se sequencialmente e simultaneamente obter velocidades de operações de dados paralelas porque o compilador trata de todos os detalhes.

Multiple Instruction Single Data trata-se de uma arquitetura pouco comum, em que vários núcleos operam no mesmo fluxo de dados através de fluxos de instruções separados.

Multiple Instruction Multiple Data refere-se a um tipo de arquitetura paralela em que múltiplos núcleos operam em vários fluxos de dados, cada um executando diferentes instruções.

Os principais objetivos a nível de arquitetura são os seguintes:

- Diminuir a latência;
- Aumentar a largura de banda;
- Aumentar a taxa de transferência.

A latência é o tempo total da operação, desde o início ao seu fim. A largura de banda é a quantidade de dados que conseguem ser processados por unidade de tempo e a taxa de transferência é a quantidade de operações que conseguem ser processadas por unidade de tempo.

As arquiteturas computacionais podem também ser subdivididas em dois tipos pela sua organização de memória, que são os seguintes:

- *Multi-node* com memória distribuída;
- *Multiprocessor* com memória partilhada.

Num sistema *multi-node*, os sistemas computacionais são construídos a partir de vários processadores conectados por rede. Cada processador tem a sua memória local e consegue comunicar os conteúdos da sua memória local pela rede. A Figura 2.14 mostra um sistema *multi-node* com memória distribuída. Estes sistemas são geralmente denominados de *clusters*.

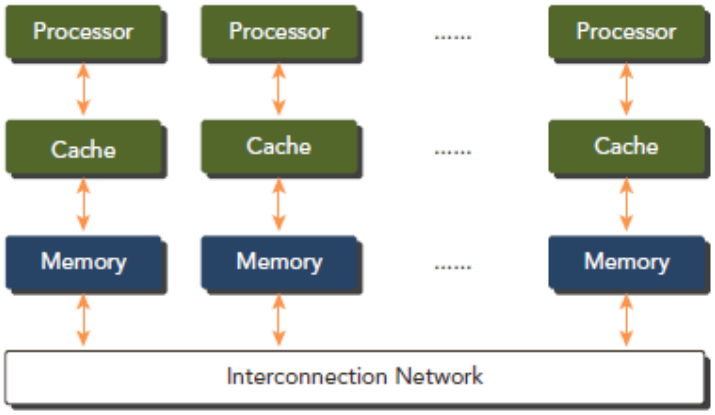

Figura 2.14: Sistema *multi-node* com memória distribuída - *cluster* [6].

As arquiteturas *multiprocessor* normalmente variam de *dual-processor* a centenas de processadores. Estes processadores ou estão fisicamente ligados à mesma memória, como pode ser visto na Figura 2.15, ou partilham uma ligação de baixa latência. Apesar de partilhar memória significar partilhar o endereço, não significa necessariamente que existe uma única memória física. Estas arquiteturas incluem tanto os sistemas de único *chip* com vários núcleos, chamado de *multicore*, como os sistemas de múltiplos *chips*, em que cada um deles tem a possibilidade de ter um *design multicore* ou *many-core*[10].

É de referenciar também as arquiteturas *Natural Processing Units* (NPU), que não são muito diferentes dos processadores *multicore* a nível organizacional. Todos os elementos de processamento são similares a núcleos comuns, contudo, partilham a mesma memória do programa, contador e registo de instruções. Todos os elementos de processamento correm a mesma instrução, o que tem impacto na programação[11].

Concluindo, os GPUs representam uma arquitetura *many-core* e têm todos os tipos de paralelismo descritos anteriormente, desde MIMD, SIMD e paralelismo a nível de instruções, o que

[10] O termo *many-core* é usado para descrever arquiteturas *multicore* com um número especialmente alto de núcleos, normalmente na casa dos milhares.

[11] https://www.cp.eng.chula.ac.th/ piak/teaching/dsys/2017/npu.htm

faz os GPUs ideais para programação paralela. A NVIDIA descreve este tipo de arquitetura como *Single Instruction, Multiple Thread* (SIMT) [36, 37].

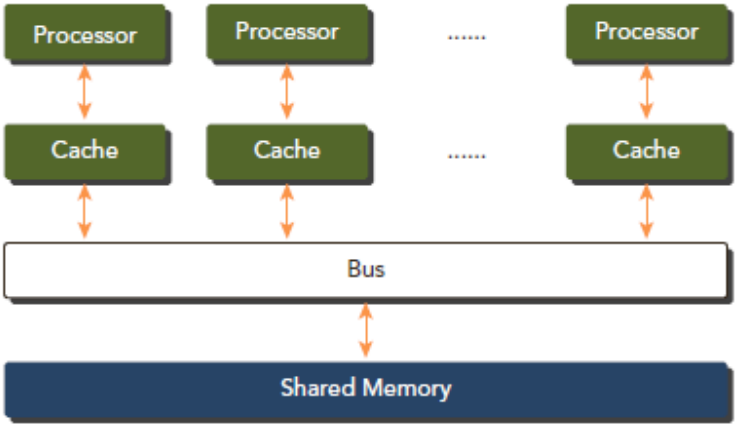

Figura 2.15: Arquitetura *multiprocessor* [6].

2.3.2 Computação Heterogénea

Nos dias embrionários dos computadores, estes apenas continham CPUs com a função de correr as mais variadas tarefas. Ultimamente, principalmente desde a última década, os computadores de HPC estão a ser alterados de modo a incluir outros elementos de processamento, como, claro está, o GPU, originalmente feito para realizar computações especializadas de gráficos paralelamente. Os GPUs tornaram-se uma ferramenta poderosa e completamente programável, capaz de realizar as mais variadas tarefas paralelamente com excelente *performance* e eficiência.

Tipicamente, os CPUs e GPUs são componentes de processamento discretas conectadas por bus PCI-Express[12](Figura 2.16). O principal aspeto que se tem de entender da computação paralela é que a computação em GPU não substitui a computação em CPU, pois o GPU é um coprocessador do CPU. Ambas se complementam e trabalham em conjunto, numa arquitetura heterogénea.

A mudança de sistemas homogéneos para sistemas heterogéneos representa um grande marco na história do HPC. A computação homogénea usa um ou mais processadores da mesma arquitetura para executar uma aplicação. A computação heterogénea, por sua vez, utiliza uma panóplia de arquiteturas de processadores para executar uma aplicação, distribuindo tarefas por arquiteturas indicadas, resultando daí uma melhoria na seu rendimento e *performance*. Estes sistemas oferecem vantagens significativas relativamente aos sistemas tradicionais de HPC, porém, o seu uso efetivo está limitado à complexidade do *design* da aplicação, e assim a inclusão de recursos heterogéneos aumenta a complexidade.

[12]PCI-Express é um padrão comunicativo utilizado para fazer a comunicação entre a placa do CPU e as restantes placas de expansão de um computador, para transmissão de dados.

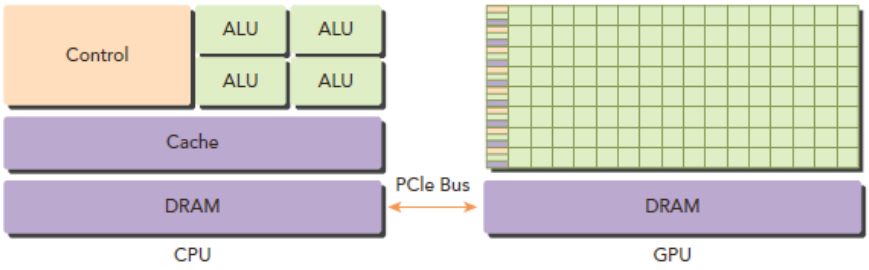

Figura 2.16: A ligação entre GPU e CPU [6].

Uma aplicação heterogénea consiste em duas partes distintas:
- Código do *host*;
- Código do *device*;

O código do *host* corre em CPUs e o código do *device* corre em GPUs. Uma aplicação executada numa plataforma heterogénea é tipicamente inicializada pelo CPU. O código do CPU é responsável por controlar o ambiente, código e informação do *device* antes de o carregar com tarefas pesadas. Os GPUs são então utilizados para acelerar as aplicações computacionalmente intensas, sendo por isso considerados como um *hardware accelerator*, ou seja, um componente de *hardware* que está fisicamente separado do CPU e é usado para acelerar secções computacionalmente intensivas de uma aplicação.

A NVIDIA, empresa criadora dos GPUs como os conhecemos em 1999 [36], tem plataformas que permitem a computação de GPU nas seguintes gamas de produtos:

- Tegra;
- GeForce;
- Quadro;
- Tesla.

Os produtos da gama da Tegra foram construídos principalmente para aparelhos móveis, como telemóveis ou *tablets*, ou como no caso desta dissertação, de plataformas robóticas móveis, como o AgRob V16 (subcapítulo 2.2.1). Os produtos da GeForce foram feitos para gráficos de consumo, os da Quadro para visualizações profissionais e os da Tesla para centros de dados de computação paralela. Os Fermi, aceleradores de GPU da gama da Tesla, ganharam recentemente uso abundante como aceleradores computacionais para aplicações HPC. Lançado em 2010, Fermi é a primeira arquitetura GPU completa, tendo já acelerado aplicações em áreas tão distintas como processamento sísmicos, simulações de bioquímica e até modelação de meteorologia [6]. A mais recente arquitetura é a Kepler, sucessora da Fermi, e foi lançada em 2012, oferecendo um poder de processamento muito maior que a antiga geração.

As duas maiores características que descrevem a capacidade de um GPU são as seguintes:

- Número de núcleos CUDA;
- Tamanho da memória.

Assim, há duas métricas que descrevem a *performance* dos GPUs:

- *Performance* computacional de pico;
- Largura de banda da memória.

Performance computacional de pico é uma medida de capacidade computacional, normalmente definida como o número de cálculos que conseguem ser processados por segundo. Banda larga da memória é a medida do rácio com que dados conseguem ser lidos ou armazenados em memória.

Para mostrar o avanço tecnológico que se consegue obter em poucos anos, na Tabela 2.4 é mostrada a comparação de *performance* das arquiteturas da Fermi e Kepler.

Tabela 2.4: Comparação entre Fermi e Kepler

	FERMI (TESLA C2050)	**KEPLER (TESLA K10)**
Núcleos CUDA	448	2 x 1536
Memória	6 GB	8 GB
Performance de pico	1.03 Tflops	4.58 Tflops
Largura de banda de memória	144 GB/s	320 GB/s

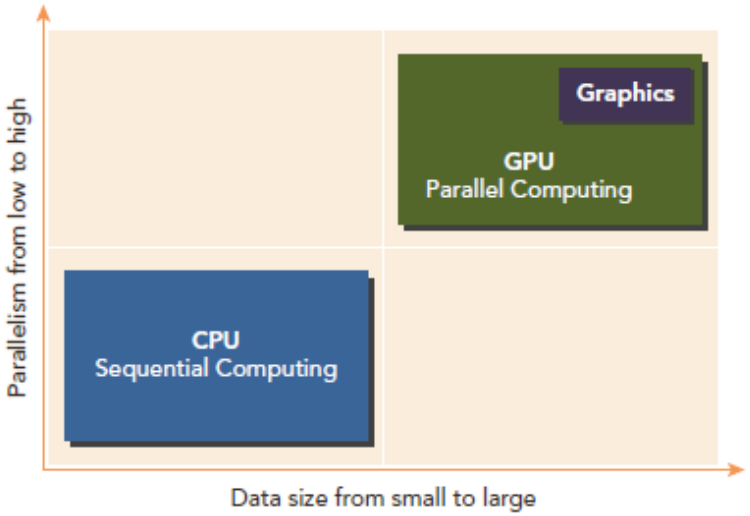

Figura 2.17: Quando utilizar CPU ou GPU [6].

A computação em GPU não tem como objetivo, tal como já foi dito, substituir a computação em CPU. Cada uma tem as suas vantagens em determinadas tarefas, como por exemplo, utilizar

CPU para tarefas dinâmicas caracterizadas por pequenas sequências de operações computacionais de controlo imprevisível, enquanto que se pode utilizar o GPU para trabalhos caracterizados por controlos simples. Na Figura 2.17 pode-se observar as duas dimensões que diferenciam a escala de aplicações para CPU e GPU:

- Nível de paralelismo;
- Tamanho dos dados.

Através da Figura 2.17 conclui-se que se um problema tem um reduzido tamanho de dados e pouco nível de paralelismo, então o CPU é uma boa escolha devido à sua capacidade de controlar lógica complexa. Se, por outro lado, o problema tem grandes quantidades de dados e mostra grande nível de paralelismo, então o GPU é o caminho a seguir, pois tem um maior número de núcleos programáveis e tem maior pico de largura de banda quando comparado com o CPU.

Contudo, um CPU também consegue fazer paralelismo a nível de instruções [38]. Cada núcleo do CPU tem grande poder computacional e com alta frequência, conseguindo executar instruções complexas, mas não muitas concorrentemente. Este limite é imposto pelo *hardware* dos seus núcleos e pelo seu número reduzido. Um GPU, pelo contrário, tem inúmeros núcleos, limitados no seu poder computacional, mas sendo vários, dependendo dos cálculos a realizar e da possibilidade do problema em causa em ser paralelizado, a arquitetura em GPU consegue atingir tempos de processamento mais elevados.

As arquiteturas paralelas heterogéneas formadas por CPU e GPU evoluíram portanto, devido aos atributos que se complementam de ambas as partes, permitindo às aplicações obter a sua melhor *performance* usando os dois tipos de processadores, ou seja, executando partes sequenciais no CPU e as partes de dados intensivos no GPU, tal como mostra a Figura 2.18.

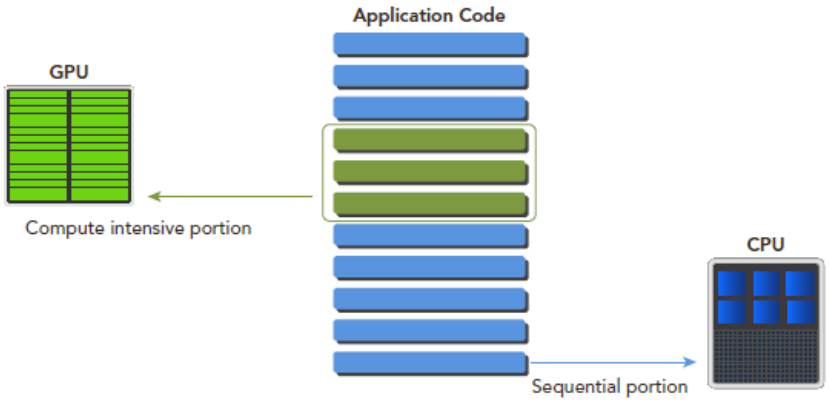

Figura 2.18: A ligação entre GPU e CPU [6].

Concluindo, programar desta forma assegura que as características do GPU e CPU se complementem, levando ao maior aproveitamento do poder e recursos do sistema. Para suportar e

encorajar o uso combinado destes processadores na programação de aplicações, a NVIDIA criou uma plataforma de programação, chamada CUDA, que será o foco do subcapítulo 2.4, enquanto que a seguir é demonstrado um teste prático de *performance* entre GPU e CPU.

2.3.2.1 Teste de *Performance*

Para se perceber as melhorias de velocidade e aceleração que os GPUs oferecem, nada melhor do que verificar isso mesmo na prática. De seguida apresenta-se uma experiência realizada em CPU, GPU e OpenMP[13] com quatro núcleos para diferentes algoritmos de processamento de imagem [8], realizada no âmbito de um estudo de *templates* para processamento em arquiteturas GPU para deteção remota de imagens. Na Tabela 2.5 pode ver-se o ambiente experimental utilizado para a experiência.

Tabela 2.5: Sistema e especificações utilizados na experiência

Sistema	**Especificações**
CPU	1×Intel(R) Xeon(R) E5504 CPU 4 Cores, 2.00GHz, L3 Cache 4 MB
Memória	4×2 GB ECC DDR3-1066
Disco	1×1 TB SATA
Ethernet	2×1000M Adaptive Ethernet Card
GPU	NVIDIA Tesla C1060
Compiladores	GCC 4.1.2, NVCC 4.0
CUDA	4.0

Os algoritmos de processamento de imagem que foram fruto de estudo desta experiência foram os seguintes:

- Transformação Logarítmica;
- Filtro Gaussiano;
- Rotação de Imagem;
- Transformada de Fourier;
- Reversão de Imagem;
- Histograma.

Estes algoritmos foram implementados com quantidade de dados crescente, nomeadamente 112 MB, 448 MB e 1.75 GB, respetivamente.

Os resultados obtidos, ou seja, o tempo de processamento dos algoritmos, em segundos, estão na Tabela 2.6.

[13] *Application Programming Interface* (API) que suporta programação multi-processo (C, C++ e Fortran), com memória partilhada para várias plataformas.

Tabela 2.6: Tempo de processamento de vários algoritmos em GPU, CPU e OpenMP [8]

	Quantidade de Dados		
Algoritmos	112 MB (s)	448 MB (s)	1.75 GB (s)
Transformação Logarítmica			
CPU	12,38	44,54	187,17
OpenMP	4,52	16,50	66,14
GPU	9,28	10,10	19,93
Filtro Gaussiano			
CPU	14,48	52,39	211,48
OpenMP	3,94	14,36	55,80
GPU	9,05	10,46	16,60
Rotação da Imagem			
CPU	13,60	56,35	253,11
OpenMP	8,45	33,34	135,35
GPU	7,82	10,39	20,92
Transformada de Fourier			
CPU	27,85	159,84	943,94
OpenMP	14,35	75,04	393,31
GPU	13,72	28,04	86,92
Reversão da Imagem			
CPU	1,98	8,33	32,74
OpenMP	1,77	7,02	27,28
GPU	16,50	36.22	64,19
Histograma			
CPU	1,78	6,01	18,43
OpenMP	1,58	4,93	15,11
GPU	12,7	21,46	30,21

Ao analisar a tabela, verifica-se que, de um modo geral, ao implementar estes algoritmos em GPU, conseguiu obter-se melhorias consideráveis na *performance* quando comparados à sua implementação em CPU ou em OpenMP. Olhando, por exemplo, para o caso da transformada de Fourier, com dados de 1.75 GB, pode observar-se que a diferença de tempos é superior a dez vezes no CPU relativamente ao GPU, uma diferença muito significativa, portanto. No caso dos algoritmos de reversão de imagem e de histograma, acontece o oposto, os tempos em GPU são bastante superiores, porém, isto pode ser explicado devido à computação destes algoritmos ser tão pequena que não se consegue usufruir do enorme poder computacional da arquitetura GPU.

Uma outra experiência estudada, uma comparação de FPGA com GPU para deteção de características de imagens de fluxo ótico em *real-time*, *stereo* e locais [39], teve como principais conclusões que a *performance* absoluta dos GPUs, quando se trata da implementação de modelos complexos, foi excecional, devido principalmente à sua grande largura de banda. Foi então con-

cluído que os GPUs, como placa coprocessadora, excedem o FPGA na maioria das comparações efetuadas, apesar de não serem indicados para muitas aplicações integradas.

2.4 Ferramentas de Programação

Esta dissertação conta com a utilização de diversas ferramentas de programação. A programação feita previamente em CPU fez uso de funções específicas de processamento de imagem, incluídas em bibliotecas *Open Source Computer Vision Library* (OpenCV), maioritariamente para visualização de resultados e apresentação de imagens. Esta programação utilizou a fácil incorporação de OpenCV com a linguagem C++ e juntou a isso a integração robótica em *Robot Operating System* (ROS).

Nos capítulos 2.4.1 e 2.4.2 são abordados aspetos gerais relativamente às ferramentas de programação mencionadas, respetivamente OpenCV e ROS. De seguida, no capítulo 2.4.3, é abordado com maior detalhe a ferramenta de programação que tem maior incidência nesta dissertação, CUDA, responsável pela programação heterogénea em GPU de placas gráficas NVIDIA.

2.4.1 OpenCV

OpenCV[14] é uma biblioteca multiplataforma *open source*, ou seja, de utilização livre para os mais variados fins académicos ou comerciais na área de visão computacional. O OpenCV possui módulos de processamento de imagem e vídeo, estrutura de dados, interface gráfica do usuário com sistema de janelas independentes e controlo com o rato e teclado. Isto para além de ter mais de 2500 algoritmos de visão computacional, desde filtros de imagem a reconhecimento de objetos, entre muitos outros. Esta biblioteca foi desenvolvida nas linguagens de programação C/C++, com suporte para C/C++, naturalmente, mas também Python ou Java em Windows, Linux, Mac OS, iOS e Android, tendo como principal foco aplicações em tempo real, como é o caso em específico desta dissertação, o algoritmo ViTruDe.

2.4.2 ROS

ROS[15] é uma coleção *open-source* de *frameworks*[16] de aplicação em plataformas robóticas. O ROS fornece serviços que são esperados de um sistema operativo, para além de ferramentas e bibliotecas para escrever e correr código em vários computadores, sendo possível a integração com código em tempo real. O ROS desenvolveu-se rapidamente e é, hoje em dia, amplamente usado pela comunidade ligada à robótica [40].

O principal objetivo do ROS é promover a reutilização de código na investigação robótica e no seu desenvolvimento. Outros objetivos incluem a fácil integração com várias linguagens de

[14] https://opencv.org/
[15] http://wiki.ros.org/
[16] Um *framework* é uma abstração que une códigos de vários projetos de *software* criando uma funcionalidade genérica.

programação, como JAVA, C++, Python, entre outras, e a fácil integração com outros *softwares* de *frameworks* robóticos.

2.4.3 CUDA

O CUDA é uma plataforma de computação paralela que tira partido da capacidade de paralelização dos GPUs da NVIDIA para resolver problemas computacionais complexos numa forma mais eficiente do que sempre foi feito tradicionalmente, em CPU, como se viu anteriormente.

A plataforma CUDA é acessível através de bibliotecas e compiladores dedicados, bem como APIs e extensões a linguagens como, por exemplo, C, C++, Fortran ou Python (Figura 2.19).

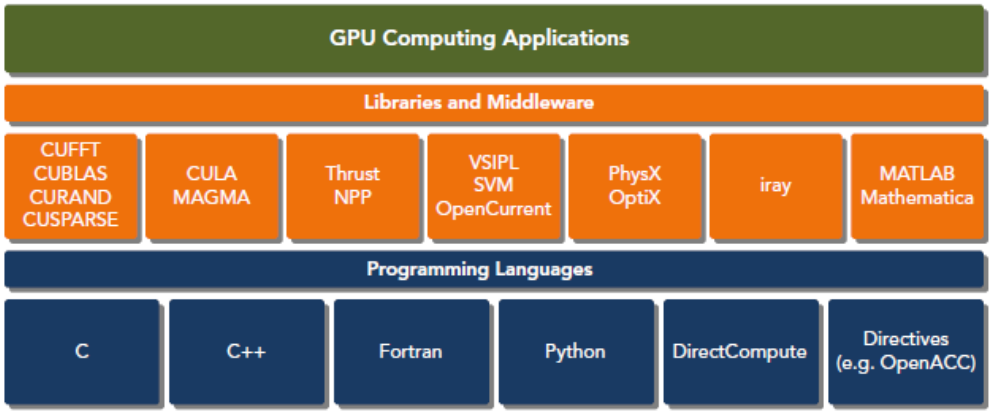

Figura 2.19: O acesso à plataforma CUDA [6].

Com vista à aceleração do algoritmo ViTruDe, será interessante o estudo da linguagem CUDA C, uma extensão do ANSI C standard com extensões de linguagem que potenciam e possibilitam a programação heterogénea. O CUDA C é também um modelo de programação escalável que permite que os programas dimensionem de forma transparente o seu paralelismo para GPUs com diferentes números de núcleos, mantendo uma curva de aprendizagem suave para programadores já familiarizados com a linguagem de programação C/C++ [6].

2.4.3.1 Programação

Um programa de CUDA consiste na junção dos dois códigos seguintes:

- Código do *host*, código que corre no CPU;
- Código do *device*, código que corre no GPU.

Durante a compilação, o compilador da NVIDIA, *nvcc*, separa o código do *device* do código do *host*. Como é demonstrado na Figura 2.20, o código do *host* é escrito em C/C++ standard e é compilado com compiladores de C/C++, enquanto que o código do *device* é escrito em CUDA C/C++, com palavras-chave que marcam as funções de paralelização, chamadas *kernels*.

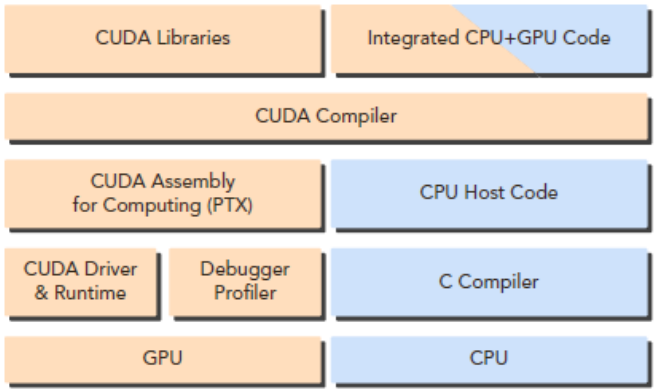

Figura 2.20: Separação do código *host* do código do *device* [6].

Um típico programa em CUDA tem a seguinte estruturação de passos a executar:

1. Alocar memórias GPU;
2. Copiar dados de memória CPU para GPU;
3. Invocar os *kernels* para processar os dados guardados em GPU;
4. Copiar de volta os dados de memória do GPU para CPU;
5. Destruir memórias alocadas em GPU.

A plataforma CUDA também suporta um ecossistema de computação paralela. Hoje em dia, cada vez mais empresas fornecem ferramentas, serviços e soluções para este tipo de computação [6]. A NVIDIA fornece o *CUDA Toolkit*[17], que inclui o compilador, bibliotecas e ferramentas para *debugging* e otimização de aplicações.

O modelo de programação CUDA funciona através de *kernels*, que com a sua chamada, consegue-se configurar o número de *threads*[18] e a forma como estas vão atuar no GPU. A chamada de um *kernel* é similar à chamada de uma função regular da linguagem C/C++ e é feita com a seguinte linha de código:

```
Kernel <<< grelha, bloco >>> (argumentos);
```

Os valores a definir para um *kernel* são os valores da grelha e do bloco. A Figura 2.21 ilustra como é composta uma grelha, que não é mais que um conjunto de blocos, em que estes, por sua vez, são conjuntos de *threads*. A dimensão da grelha refere-se ao número de blocos GPU a serem lançados, enquanto que a dimensão do bloco dá a indicação de quantas *threads* são lançadas por cada bloco.

As *threads* referentes ao mesmo bloco conseguem facilmente comunicar entre si, porém, *threads* de blocos diferentes não conseguem cooperar.

[17] https://developer.nvidia.com/cuda-toolkit
[18] Cada *thread* representa uma execução singular de um dado *kernel*.

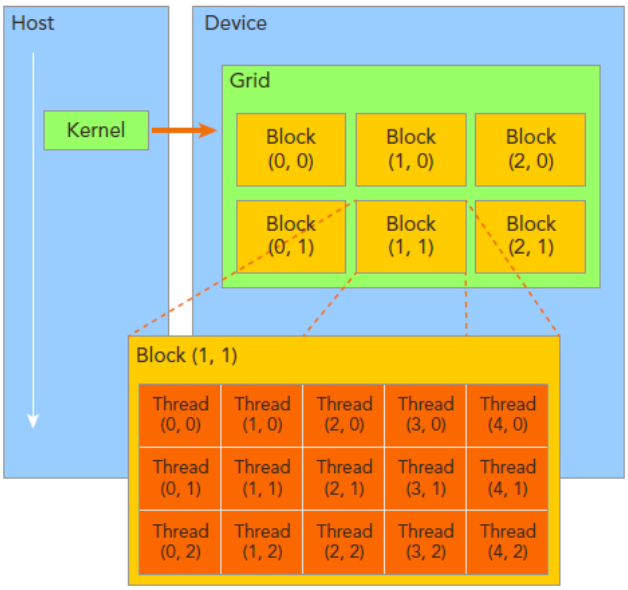

Figura 2.21: Relação hierárquica entre grelhas, blocos e *threads* [6].

Para um dado problema, usando diferentes configurações de dimensões de grelha e de bloco é possível organizar todo o *layout* das *threads*. Por exemplo, supondo que se tem 32 elementos de dados que se pretende que sejam paralelizados e que não é necessária a passagem de argumentos adicionais. Uma das inúmeras alternativas seria agrupar oito elementos por cada bloco e lançar quatro blocos de oito *threads* cada, de modo a que cada *thread* consiga realizar as operações de um dado elemento, totalizando os trinta e dois. Esta chamada do *kernel* seria então feita da seguinte forma:

```
Kernel <<< 4, 8 >>> ();
```

A organização das *threads* deste *kernel* está ilustrada na Figura 2.22. Cada *thread* lançada vai significar uma execução concorrente do código do *kernel*. Cada *thread* possui uma variável identificadora única, tal como cada bloco, *threadIdx.x* e *blockIdx.x*, respetivamente.

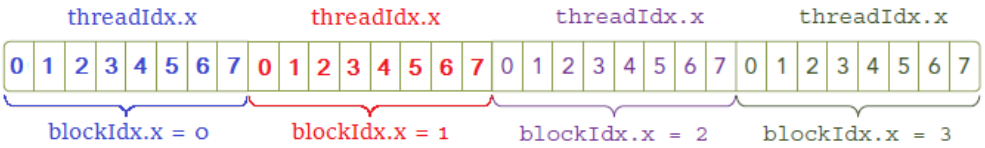

Figura 2.22: Número de execuções concorrentes lançadas por quatro blocos de oito *threads* cada.

O código do *device* é assíncrono relativamente ao código do *host*. Quer isto dizer que após o CPU lançar o *kernel*, o controlo retorna imediatamente ao CPU. Os códigos *host* e *device* conseguem correr assim simultaneamente.

Como foi assinalado, existem inúmeras opções e configurações para uma correta execução de um *kernel* para um problema em concreto. Permitindo o controlo ao utilizador da organização de

threads que vão executar o problema, o CUDA oferece assim uma capacidade de paralelização de dados massiva [6].

2.4.3.2 Execução

Na teoria, todas as *threads* correm de forma paralela dentro de um bloco, mas a realidade é que fisicamente, do ponto de vista do *hardware*, isso é impossível. Dentro do mesmo bloco, *threads* diferentes avançam a um ritmo diferente entre si, por isso, partilhar os mesmos dados entre *threads* paralelas pode originar uma condição de corrida: várias *threads* acedendo ao mesmo endereço numa ordem não definida, o que pode levar a resultados imprevisíveis por parte do programa.

Warps são a unidade básica de execução num *Streaming Multiprocessor*[19] (SM). Quando é lançada uma grelha de blocos, esses blocos são distribuídos pelos SMs. A partir de um momento que um bloco é atribuído a um SM, os seus *threads* são partidos em *warps*. Um *warp* consiste em 32 *threads* consecutivos e todos os *threads* de um *warp* são executados em *Single Instruction Multiple Thread* (SIMT - ver secção referente às arquiteturas computacionais, 2.3.1), ou seja, todos os *threads* executam a mesma instrução, mas cada *thread* executa a sua operação de forma privada, o que basicamente significa que, por exemplo, uma variável declarada dentro da execução de um *thread*, será declarada apenas e só para utilização desse *thread*.

Deste modo, a execução de programas que utilizem o máximo da capacidade dos *warps*, como por exemplo utilizar quantidades de *threads* que sejam múltiplas da dimensão do *warp*, permitirá que se utilizem todos os recursos disponíveis, aumentando por isso a *performance* global do *kernel*.

2.4.3.3 Memória

Na Figura 2.23 pode observar-se os diferentes tipos de memória que se têm de ter em conta na programação CUDA, ordenados pelo seu tamanho e velocidade de acesso. Acessos pobres aos diferentes tipos de memória podem causar grandes perdas de *performance*.

A gestão dos acessos à memória é parte integral de qualquer linguagem de programação. Muitos sistemas são limitados pela rapidez com que conseguem guardar e carregar dados, por isso, ter uma memória de baixa latência e grande largura de banda é naturalmente benéfica para o rendimento do sistema. Contudo, memórias de grande capacidade e de alta *performance* nem sempre são possíveis de atingir, nem tão pouco economicamente viáveis. Assim, tendo em conta o *hardware*, tem de se olhar ao modelo de memória para se conseguir atingir a latência e largura de banda ótimas.

Tanto os CPUs como os GPUs usam os mesmos princípios e modelos na hierarquia de memória, a diferença é que o modelo de programação CUDA oferece controlo mais explícito sobre a sua memória [6].

[19] Número de *Streaming Processors* - SPs, ou seja, núcleos CUDA. Cada bloco de *threads* é atribuído a um SM e o seu número informa, então, acerca do número de blocos que conseguem ser executados paralelamente.

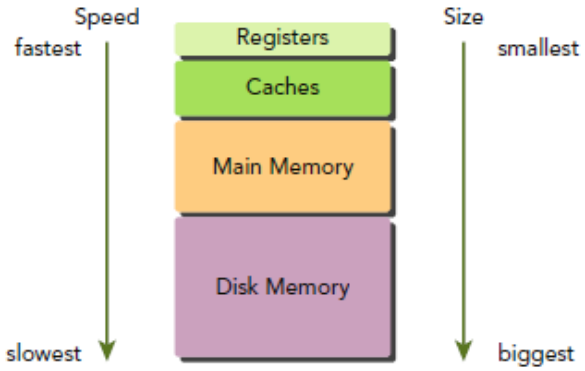

Figura 2.23: Organização de memórias relativamente à sua dimensão e velocidade [6].

Existem dois tipos de memória:

- Programável - Existe controlo explícito sobre os dados guardados neste tipo de memória;
- Não Programável - Não existe controlo sobre a forma como são guardados os dados.

Os diferentes tipos de memória são abordados em maior detalhe de seguida. Na Figura 2.24, a laranja, estão representadas as memórias programáveis em CUDA e as suas interações dentro das grelhas e dos blocos.

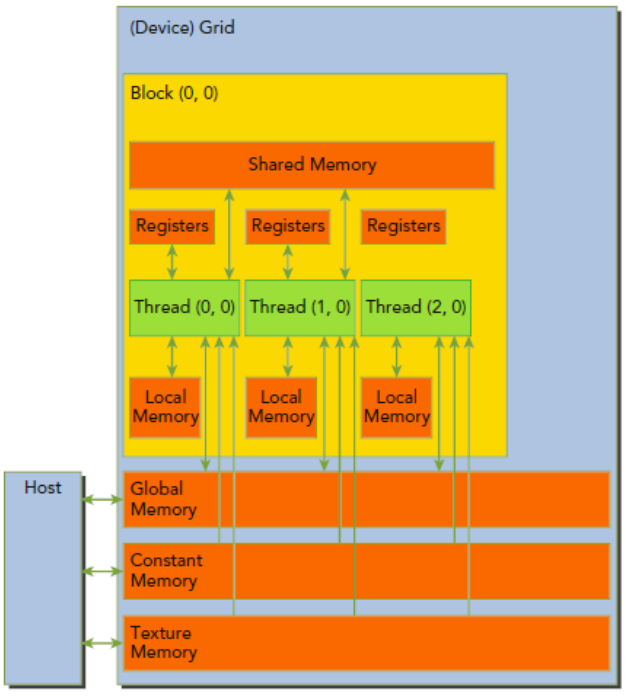

Figura 2.24: Memórias programáveis em CUDA [6].

Registos:

Os registos são os espaços de memória mais rápidos num GPU. Qualquer variável declarada num *kernel*, sem qualquer tipo de qualificadores que apontem a outra memória, é guardada num registo. As variáveis guardadas em registos são privadas para cada *thread* e quando o *kernel* acaba a sua execução estas variáveis não podem ser novamente acedidas visto que os registos são limpos.

Memória Local:

Variáveis que são elegíveis para os registos mas que ultrapassam as suas dimensões são guardadas na memória local. O termo "local" é enganoso, visto que esta memória reside no mesmo espaço físico que a memória global, e, como se pode ver na Figura 2.23, os acessos a esta memória são caracterizados por alta latência e baixa largura de banda.

Memória Partilhada:

A localização física deste tipo de memória faz com que tenha uma largura de banda muito superior e uma latência muito inferior relativamente à memória local ou global. Cada SM tem uma quantidade limitada de memória partilhada que é dividida pelos blocos, por isso este tipo de memória é facilmente preenchido se for sobreutilizado. Esta memória é libertada aquando da finalização da execução do bloco ao qual foi alocada.

Memória Constante:

A memória constante é armazenada numa *cache* dedicada por cada SM. A dimensão máxima de declaração deste tipo de memória é de 64 kB, sendo obrigatoriamente uma declaração estática, ou seja, uma variável deste tipo será apenas de leitura e é visível para todos os *kernels* existentes no programa.

Memória de Textura:

A memória de textura é uma memória global acedida por uma *cache* dedicada de apenas leitura, por cada SM. Este tipo de memória é otimizado para localização espacial 2D, por isso *threads* que façam processamento em dados 2D terão *performance* superior utilizando esta memória em detrimento da memória global.

Caches do GPU:

Tal como *caches* em CPU, este tipo de *caches* são espaços não programáveis de memória. Existem quatro tipos de *cache* em GPU: L1, L2, *cache* da memória constante e *cache* da memória de textura.

Memória Global:

A memória global é o maior tipo de memória, porém com os tempos de acesso mais lentos. Esta memória pode ser acedida por qualquer SM durante o tempo de vida de toda a aplicação. Uma variável declarada neste tipo de memória pode ser estática ou dinâmica e como a execução de *threads* pelos diferentes blocos não pode ser sincronizada, várias *threads* a tentar modificar o valor do mesmo endereço da memória global pode causar comportamentos não desejados. Ao acederem a endereços de memória, as *threads* bloqueiam até esses acessos estarem concluídos, e desse modo é agendado para execução outro *warp*, e sempre assim até todos os acessos estarem realizados. Este fenómeno chama-se *latency hiding*, e devido a ele, é importante lançar uma execução de *kernel* com muitos *warps*, de forma a manter todos os núcleos da GPU ocupados e

maximizar assim os recursos.

Otimizar os acessos a este tipo de memória é vital para uma boa *performance*, e portanto, tem de se fazer bom uso de funções que transferem dados do *host* para o *device*, e vice-versa.

Devido à natureza dos acessos à memória em CUDA, realizados em *warps*, para se obter a melhor *performance* na leitura e escrita de dados, os acessos à memória devem ser alinhados e unidos/adjacentes. A Figura 2.25 ilustra um acesso à memória alinhado e unido.

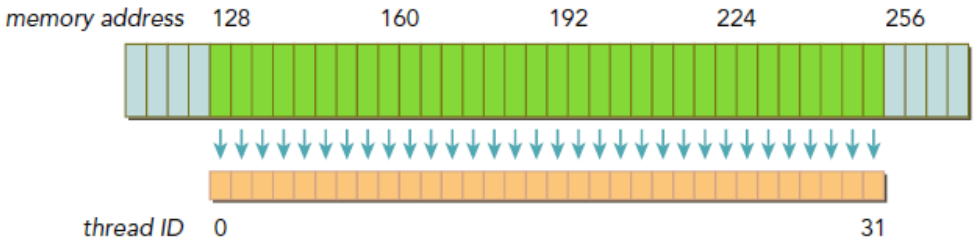

Figura 2.25: Exemplo de um acesso à memória alinhado e unido [6].

Um acesso alinhado ocorre quando o primeiro endereço da transação de memória é um múltiplo par da *cache* que executa a transação (32 bytes para o L2 e 128 bytes para o L1), enquanto que um acesso unido ocorre quando os 32 *threads* dum *warp* acedem um pedaço de memória adjacente. Assim, para maximizar a *performance* e não desperdiçar largura de banda, os acessos à memória devem ser feitos a endereços de memória contíguos e iniciados num endereço alinhado. Desta forma, é realizada uma única transação de memória (os valores de cada endereço são enviados para as *threads* por *broadcast*) em vez de ter acessos não alinhados nem adjacentes, que faz com que haja várias transações de memória realizadas em série. É relativamente fácil conseguir alinhar os acessos à memória, mas alguns algoritmos podem fazer os acessos unidos/adjacentes difíceis de atingir ou até mesmo impossíveis [6].

Assim, deve-se otimizar as transações de memória utilizando o menor número de transações possível para o máximo número de solicitações de memória.

Em suma, a plataforma CUDA mostra ser a ferramenta mais indicada para fazer a paralelização dos elementos de perceção visual necessários ao ViTruDe, permitindo, através da sua computação heterogénea, providenciar melhorias significativas na *performance* deste algoritmo.

Capítulo 3

ViTruDe - Abordagem em GPU

Neste capítulo será realizada a caracterização do problema e tudo aquilo que foi realizado para se conseguir chegar aos resultados desejados.

Na secção 3.1 é feita a caracterização geral do problema que esta dissertação retrata. No tópico 3.1.1 são propostos os trabalhos a efetuar para solucionar o problema em causa.

Na secção 3.2 são descritos os passos iniciais da implementação, nomeadamente as configurações do ambiente de trabalho que possibilitam a utilização em conjunto das ferramentas aplicadas.

De seguida, na secção 3.3 é detalhado o que foi realizado para treinar e testar a SVM para a correta classificação dos *pixels* nas classes consideradas, bem como a análise ao modelo de treino resultante.

Finalmente, na secção 3.4 é exposta a abordagem ao problema em concreto, ou seja, são clarificados os algoritmos e códigos implementados da programação paralela do ViTruDe de maneira a se conseguir obter resultados satisfatórios. No tópico 3.4.1 são expostas algumas alterações ao algoritmo apresentado de forma a obter-se uma maior otimização da paralelização apresentada.

3.1 Caracterização do Problema

Um dos grandes desafios atuais da robótica para agricultura é alcançar algoritmos de processamento de imagem que sejam robustos a todas as condições de luminosidade, ao mesmo tempo que são eficientes e com custos adequados ao contexto em que se encontram.

O projeto ROMOVI tem por base a criação de uma plataforma robótica capaz de se enquadrar no plano de agricultura de vinhas de montanha, nomeadamente as vinhas das margens do rio Douro, e providenciar uma ajuda extra a esse setor nessa região, em particular oferecendo ajuda com a monitorização e colheita, por exemplo.

De modo a que a plataforma robótica consiga realizar eficientemente estas tarefas, necessita de uma estimativa precisa de localização, cuja correta implementação tem por base algoritmos de deteção de elementos naturais das vinhas, como é o caso do algoritmo ViTruDe. Este algoritmo, que já havia sido anteriormente desenvolvido [41], é computacionalmente pesado e apresenta um processamento das imagens captadas muito demorado, necessitando de cerca de 10 minutos para

uma análise a uma *frame* completa.

O que se pretende com esta dissertação é tornar este processamento do ViTruDe mais rápido, de modo a viabilizar a sua utilização em tempo real pelo AgRob V16.

3.1.1 Solução Proposta

Como foi referido, a presente dissertação tem como principal objetivo a aceleração de algoritmos de perceção visual de troncos de vinhas de encosta, anteriormente desenvolvidos no projeto ROMOVI, em CPU [41].

A solução proposta desta dissertação prende-se com uma implementação em contexto de GPUs, fazendo uso da sua capacidade de programação heterogénea para acelerar o processo da deteção dos elementos naturais das vinhas. Será necessário decompor os algoritmos de perceção visual num conjunto de funções que possam ser executadas de forma concorrente e desenvolver um conjunto de bibliotecas dedicadas.

A abordagem apresentada passa por fazer uso da plataforma robótica existente e dos seus dados previamente adquiridos, bem como contar com o OpenCV, ROS e a plataforma CUDA, da NVIDIA, como ferramentas para a correta modelização e programação paralela do algoritmo ViTruDe. É esperado que o tempo de execução deste processamento de imagem diminua significativamente e que torne a implementação em GPU deste algoritmo mais viável para utilização em tempo real quando comparada com a sua prática em CPU.

3.2 Introdução

O primeiro passo para a realização e implementação da dissertação foi a execução dos algoritmos ViTruDe em CPU [41], de maneira a ser possível observar e analisar não só uma demonstração prática da deteção dos elementos naturais das vinhas em causa neste projeto, como também fazer uma prospeção acerca do tempo de processamento total despendido nessa abordagem. As versões mais rápidas do ViTruDe são aquelas em que primeiramente são detetados pontos-chave na imagem e só depois é feito o processamento apenas nesses pontos, o que permite poupar no tempo de execução, mas não apresenta, obviamente, resultados tão fiáveis como um processamento a todos os *pixels* da imagem. A versão do ViTruDe mais demorada, mas com maior precisão, é por isso aquela que processa todos os *pixels* de cada imagem de entrada, podendo demorar intervalos de tempo incrivelmente longos na implementação num CPU Intel® Core™ i7-4700HQ, como cerca de 10 minutos para uma única *frame* de vídeo.

De seguida, foi necessário a configuração do ambiente de trabalho. Visto que a NVIDIA oferece um *Integrated Development Environment* (IDE) com funcionalidades próprias dedicadas à programação CUDA, o NSight, o objetivo passou por tirar partido desse IDE, tendo em consideração que o projeto é operado com recurso ao ROS. Nesse sentido foram realizadas todas as alterações e adições que permitissem a correta compilação e execução do algoritmo, juntando as duas vertentes. No apêndice A estão explicadas em maior detalhe as configurações realizadas, e,

para *benchmarking*[1] e auxílio de futuros trabalhos relacionados com programação heterogénea, está transcrito nesse apêndice, na sua totalidade, o ficheiro CMakeLists.txt deste projeto, responsável pela compilação de pacotes ROS.

A implementação dos algoritmos foi realizada de forma faseada e aplicada numa imagem de um *dataset* real, captado pela câmara do AgRob e disponibilizado publicamente[2] (Figura 3.1).

De notar também que, primeiramente, todos os procedimentos foram realizados num computador com processador CPU Intel® Core™ i7-4700HQ, e GPU NVIDIA GeForce GT 740M[3], e posteriormente foi testada a GPU NVIDIA GeForce GTX 1050 TI[4]. As principais diferenças entre as propriedades para programação CUDA destas GPUs com a GPU do robot, NVIDIA Tegra K1, estão representadas na Tabela B.1 do apêndice B.

Figura 3.1: Imagem utilizada para desenvolvimento dos algoritmos.

3.3 Modelo de Treino

Os resultados em CPU do ViTruDe [41] que obtiveram maior precisão na deteção de troncos foram aqueles que utilizaram descritores do tipo *plus colour*, SVM linear e de quatro classes, porém, utilizar quatro classes implica maior carga de trabalhos e prejudica substancialmente o tempo de processamento.

Assim, e como esta dissertação tem como o objetivo uma maior eficiência em relação ao tempo e à velocidade de execução, decidiu-se utilizar as configurações da SVM que permitissem isso mesmo. Portanto, a SVM a utilizar é linear e recorre a apenas duas classes de classificação, "Tronco" e "Não Tronco". Isto permitirá tempos de processamento bastante mais rápidos, aproximando o ViTruDe de um algoritmo de execução em tempo real. Estabelecidos estes parâmetros, o treino e teste da SVM foi efetuado com recurso ao *Library for Support Vector Machines* (libSVM[5])

[1] O *benchmarking* analisa estratégias e possibilita a criação de ideias novas posteriormente ao que já foi realizado.
[2] *Datasets* retirados de: http://vcriis01.inesctec.pt/datasets/DataSet/AGROB/
[3] https://www.geforce.com/hardware/notebook-gpus/geforce-gt-740m
[4] https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1050/
[5] Aplicação *open source* em CPU de uma SVM [42], utilizada no cálculo do ViTruDe em CPU [41].

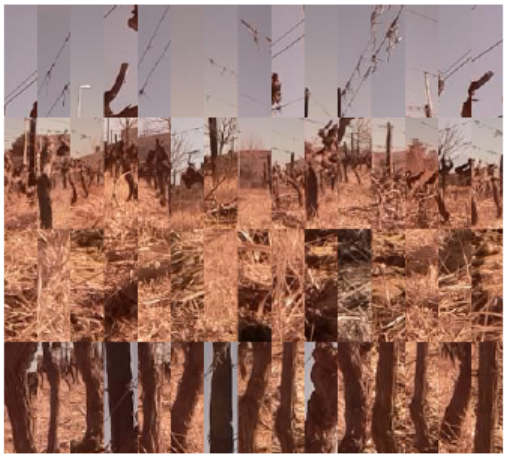

Figura 3.2: Amostras de imagens do *dataset* utilizado.

para um *dataset* de 2000 imagens reais, de treino e teste, obtidas nas videiras na Universidade de Trás-os-Montes e Alto Douro (UTAD), de manhã, com uma câmara sem infravermelhos nem filtro azul. As imagens "Tronco" e "Não Tronco" do *dataset* utilizado para o treino estão representadas na Figura 3.2 e, após teste, obteve-se a matriz de confusão representada na Tabela 3.1.

Tabela 3.1: Representação da matriz de confusão obtida após treino e teste da SVM

Matriz de Confusão	Tronco	Não tronco
Tronco	498	2
Não tronco	0	1500

Analisando a matriz de confusão verifica-se que o treino foi realizado com sucesso, pois para os 500 testes de imagens de tronco realizados, 498 tiveram sucesso, e para os 1500 testes de imagens de não tronco (que incluem céu, folhas e chão), houve 100% de taxa de sucesso na sua classificação como não sendo tronco.

Como forma de comparação, neste modelo de treino resultante constam 128 *Support Vectors* (SVs) de 264 posições cada e 128 coeficientes, números muito reduzidos quando comparados com, por exemplo, um treino de núcleo polinomial de quatro classes, que inclui 1370 SVs.

O objetivo é que o treino seja feito apenas uma vez, ou seja, a partir do momento que se obteve este modelo de treino, ser possível aceder diretamente aos seus parâmetros de modo a que estes possam ser utilizados de forma adequada na classificação da imagem.

3.4 Programação Paralela do ViTruDe

O que se pretende é paralelizar o processamento de todos os *pixels* da imagem. Em CPU, foram adicionadas medidas que permitissem contornar a necessidade de processar todos os *pixels* para diminuir o tempo de execução, como a deteção de pontos-chave (ver subcapítulo 2.2 do

capítulo 2 - revisão bibliográfica) e um processamento exclusivo a esses pontos-chave. Porém, em GPU, o objetivo passa por restringir o menos possível a análise de dados, tendo sempre em consideração a tentativa de obtenção do menor tempo de processamento, pois esse é o principal objetivo desta dissertação.

O processamento de imagem do ViTruDe pode ser decomposto nas seguintes funções:

- Leitura da imagem de entrada;
- Extração da imagem LBP a partir da imagem de entrada;
- Extração da imagem HSV a partir da imagem de entrada;
- Cálculo dos descritores em cada *pixel*;
- Normalização dos descritores em cada *pixel*;
- Classificação dos descritores em cada *pixel*.

Estas diferentes funções foram então analisadas para aplicação à imagem da Figura 3.1, de forma a determinar o que é possível paralelizar em cada uma dessas fases e até que nível é vantajoso utilizar o recurso a GPU para as realizar.

Leitura da imagem de entrada:

O contexto da utilização do OpenCV nesta dissertação prende-se, de forma exclusiva, com a leitura de imagens e com a amostragem e visualização de resultados, tal como havia sido feito na aplicação CPU do ViTruDe [41].

Deste modo, a abertura, leitura e redimensionamento (para o tamanho pré-definido de 640x480) de cada imagem é realizada numa simples operação em CPU disponibilizada pelas bibliotecas de OpenCV dedicadas ao processamento de imagem. Aqui, o conjunto de *pixels* da intensidade da imagem são guardados numa matriz definida por estas bibliotecas, matriz esta que devido aos problemas de incompatibilidade não pode ser tratada dentro de um *kernel* CUDA, por isso, para haver a possibilidade de passar a imagem como parâmetro para uma função GPU deste tipo para que seja processada no seu interior, tem de se converter primeiro a matriz num vetor comum da linguagem C, como num vetor de caracteres sem sinal, de tamanho $Linhas \times 640 + Colunas$ (ou $Colunas \times 480 + Linhas$).

Extração da imagem LBP:

Estando a imagem lida e convertida num formato que possibilita a leitura de funções CUDA, o próximo passo foi obter a imagem LBP (Figura 3.4) a partir da imagem de entrada.

Nesta fase, como foi explicado em 2.2.2, após ser convertida para escala-cinza, a imagem de entrada foi analisada *pixel* a *pixel* e comparada a sua intensidade com a da vizinhança, num raio de

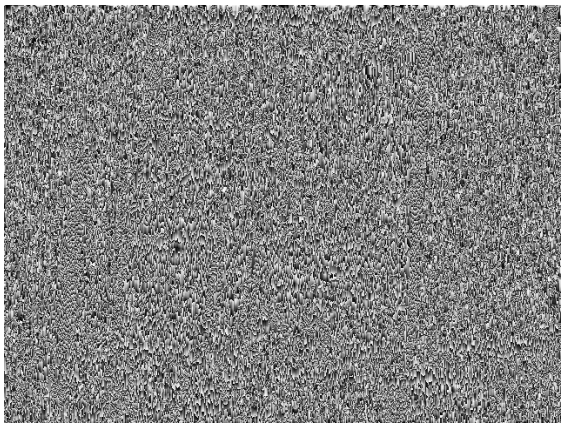

Figura 3.3: Imagem LBP ideal.

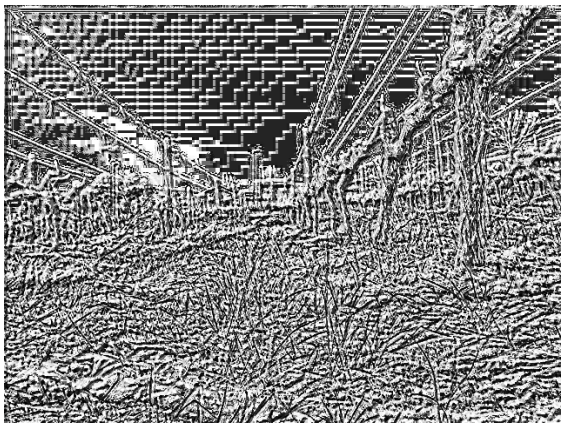

Figura 3.4: Imagem LBP extraída em GPU.

3x3 *pixels*. Isto vai permitir a extração de um código binário de padrões - código LBP. Esta fase tem um elevado grau de paralelismo, na medida em que é possível processar uma grande gama de *pixels* concorrentemente, ao contrário da sua implementação em CPU, que apenas permite o processamento de um *pixel* de cada vez, de forma sequencial, tornando o processo naturalmente mais demorado.

Assim, foi executado um *kernel* que permita que 640 *threads* atuem simultaneamente sobre a imagem, cada uma analisando uma coluna da imagem (480 *pixels*).

Na listagem 3.1 é apresentado um pseudo-código que ilustra o que cada uma das 640 *threads* vai efetuar para obter o código LBP:

Listagem 3.1: Obtenção do Código LBP

```
Ciclo (480)
    codigo_LBP = 0;

    Se ( imagem[pixel_atual] <= imagem[1_vizinho])
        codigo_LBP += 1;
    Se ( imagem[pixel_atual] <= imagem[2_vizinho])
      codigo_LBP += 2;
    Se ( imagem[pixel_atual] <= imagem[3_vizinho])
      codigo_LBP += 4;
    Se ( imagem[pixel_atual] <= imagem[4_vizinho])
      codigo_LBP += 8;
    Se ( imagem[pixel_atual] <= imagem[5_vizinho])
      codigo_LBP += 16;
    Se ( imagem[pixel_atual] <= imagem[6_vizinho])
      codigo_LBP += 32;
    Se ( imagem[pixel_atual] <= imagem[7_vizinho])
      codigo_LBP += 64;
```

```
Se ( imagem[pixel_atual] <= imagem[8_vizinho])
  codigo_LBP += 128;

Saida[pixel_atual]=codigo_LBP;
```

No final deste *kernel* em GPU, fica-se com um código binário LBP para cada *pixel*, obtendo assim a figura LBP pretendida, porém, a imagem LBP ideal seria a da Figura 3.3, pois é aquela que teoricamente permitiria obter melhores resultados ao nível do sucesso na deteção de elementos naturais das vinhas. Contudo, para que esta operação possa ser realizada em GPU, o código teve de ser ligeiramente alterado, porque em CPU, como cada *pixel* é analisado de forma sequencial, o valor "codigo_LBP" é guardado para utilização na iteração seguinte, ou seja, cada processamento de um *pixel* necessita do valor do processamento do *pixel* anterior, o que simplesmente não é possível quando se está a analisar todos os *pixels* de forma concorrente. Concluiu-se então que, ao realizar várias iterações de forma paralela, uma iteração não consegue ir buscar um valor resultante de outra iteração que está a correr paralelamente a si, por isso tiveram de ser feitas alterações no algoritmo que das quais resultou a imagem da Figura 3.4, que é uma versão aproximada à imagem LBP ideal, mas programada em GPU e com o que isso implica.

Este *kernel* é ótimo no sentido da aceleração por GPU, pois não obriga as *threads* a um cálculo demasiado complexo (apenas verificações de condições e incremento de uma variável de registo) e os acessos aos endereços de memória são todos unidos e alinhados, por isso, pode-se considerar este *kernel* como perfeito para uma implementação em programação paralela. Os resultados obtidos relativamente ao tempo de processamento podem ser visualizados no capítulo 4.

Extração da imagem HSV:

Simultaneamente à extração da imagem LBP, é extraída a imagem HSV a partir da imagem original (Figura 3.1), em CPU. Esta fase de processamento não é feita em GPU, pois, com o auxílio das bibliotecas OpenCV dedicadas, a obtenção da imagem HSV fica bastante facilitada.

Esta operação consegue, mesmo assim, ser realizada paralelamente à extração da imagem LBP, pois o código *host*, ao chamar o *kernel* de extração da imagem LBP, continua a sua execução e não espera que esse *kernel* termine, a menos que essa condição seja imposta.

Tendo a imagem HSV (Figura 3.5), guarda-se cada uma das suas três dimensões em três vetores de caracteres sem sinal. Um vetor para a componente "Hue", outro para a "Saturation" e outro para a componente "Value".

No algoritmo ViTruDe, a componente que terá aplicação é a "Hue", ou seja, a componente das gamas de cor. Esta componente será reduzida a oito espaços de cor. A listagem 3.2 representa, em pseudo-código, o procedimento efetuado.

Figura 3.5: Imagem HSV obtida a partir da imagem de entrada.

Listagem 3.2: Obtenção das gamas de cor

```
Ler todos os pixels da imagem HSV, e, para cada um:
    Hue = pixel[0]/32;
    Saturation = pixel[1];
    Value = pixel[2];

    Se (Value demasiado pequeno)
        Hue = 0;
    Se (Saturation demasiado pequeno)
        Hue = 1;
    Se (Hue > 8)
        Erro;

    Gama_de_cor[pixel] = Hue;
```

O que vai resultar desta fase de processamento é um vetor de dimensão $640 \times 480 = 307200$ posições, em que cada posição sua terá um valor variante na escala [0,7], representativa das oito gamas de cor.

Cálculo dos descritores:

Depois de serem extraídos os códigos LBP a partir da imagem LBP e as gamas de cor a partir da imagem HSV, o próximo passo foi o cálculo dos descritores. Este cálculo é integralmente feito em GPU de forma paralelizada e para isso recorre-se a um *kernel* CUDA, que é responsável pela sua inicialização e incrementação.

O descritor mais eficiente é aquele que é utilizado nesta fase - o *hLBP plus Colour* - e pode

ser visto como um histograma de 264 colunas, tal como representado abaixo na Figura 3.6.

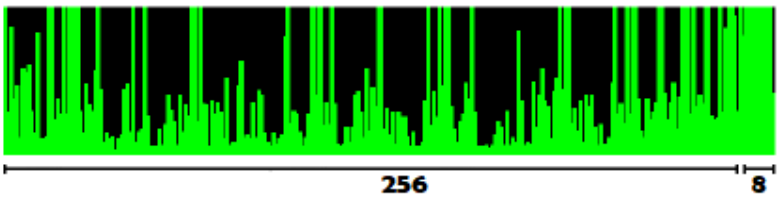

Figura 3.6: Exemplo de descritor ilustrado em forma de histograma.

As primeiras 256 colunas desse histograma contêm o número de códigos LBP, enquanto que as últimas 8 colunas contêm o número de gamas de cor, sendo que, para este descritor foi selecionado o LBP com 8 gamas no espaço de cor. Assim resulta um histograma de LBP de tamanho 2^8 (ou seja, 256) , e o histograma final terá então 264 colunas (256 + 8).

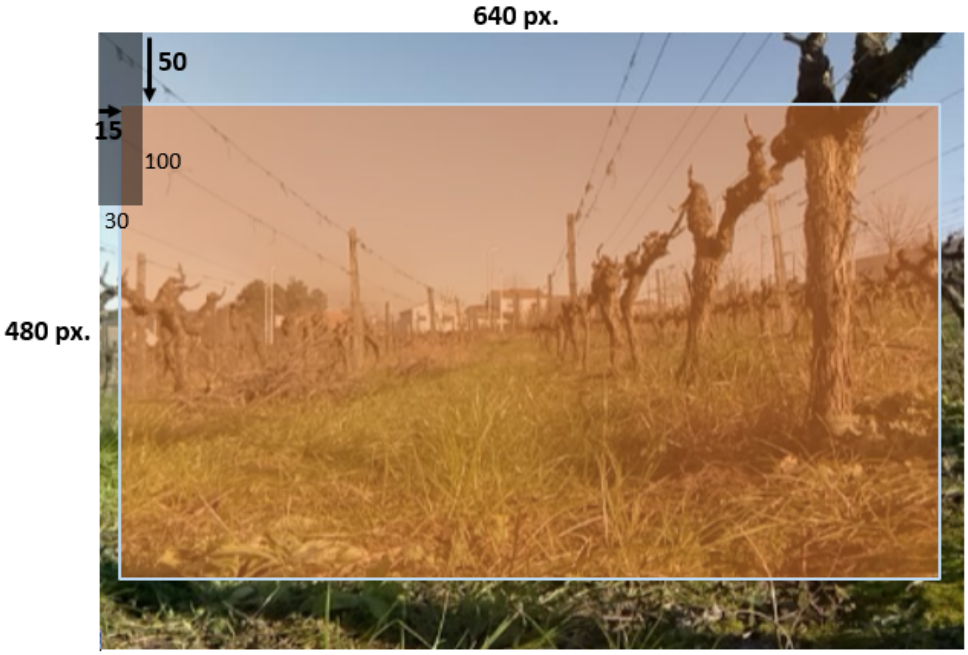

Figura 3.7: Ilustração das janelas analisadas.

Para o cálculo em GPU, o primeiro passo foi a declaração e inicialização de todos os valores de todos os descritores a zero. Depois, como se pode observar na Figura 3.7 (não está à escala), a imagem será percorrida por uma janela de análise 30x100, que se centra no *pixel* a ser analisado - o *pixel* central - e é feita uma leitura dos códigos LBP e das gamas de cor dentro dessa janela, de modo a preencher cada coluna do histograma com o número de leituras efetuadas para cada valor. As imagens LBP e de gamas de cor serão percorridas com esta janela e daí vai resultar uma zona de 610x380 *pixels* para classificação, portanto, o número total de descritores é dado por $610 \times 380 = 231800$ descritores, representados pela cor alaranjada na Figura 3.7.

Na abordagem em GPU, para fazer o cálculo de todos os descritores, são lançadas 610 *threads*, e cada uma dessas *threads* tem a função de percorrer uma coluna das imagens LBP de gamas de cor, analisando a janela 30x100 para cada um dos 380 *pixels* da respetiva coluna, incrementando as posições de cada descritor, ou, visto de outra perspetiva, incrementando as colunas de cada histograma consoante o valor encontrado nas imagens LBP e de gamas de cor.

Por exemplo, se, para um dado *pixel*, uma *thread* encontrar o valor "150" na imagem LBP e "2" na imagem de gamas de cor, essa *thread* vai incrementar com uma unidade as colunas 150 e 258. Se na janela 30x100 a ser analisada pela *thread* forem encontrados cinquenta códigos LBP com o valor de "150", então a coluna 150 do histograma terá um dimensão de 50.

Fazendo as contas (Equação 3.1), começa-se a perceber o porquê deste método de perceção visual ser uma operação tão demorada - Para cada um dos 231800 descritores são analisados 3000 *pixels* duas vezes (janela de 30x100 para imagem LBP e outra janela de iguais dimensões para imagem de gamas de cor).

$$\begin{aligned} \textit{Pixels} \text{ totais analisados} &= (2 \times 3000) \times 231800 \\ &\Rightarrow \textit{Pixels} \text{ totais analisados} = 1\ 390\ 800\ 000 \end{aligned} \tag{3.1}$$

O número de *pixels* totais analisados será então 1 390 800 000, um número astronómico, que espelha bem a quantidade de análise de dados relacionada com este tipo de processamento de imagem.

Este *kernel* obriga as *threads* a uma complexa computação e, para além disso, os acessos aos endereços de memória por parte de *threads* do mesmo *warp* não são os melhores, na medida em que as colunas dos histogramas a serem incrementadas sofrem variação de *pixel* para *pixel* e, portanto, os endereços de memória acedidos raramente serão adjacentes. É expectável, então, que o algoritmo nesta fase não tenha a eficiência das fases anteriores em termos de tempo de processamento.

Dito isto, neste momento já se tem 231800 descritores de 264 posições cada, sendo necessária a sua classificação. Porém, é necessário um passo extra para que a classificação tenha sucesso: a normalização dos $231800 \times 264 = 61\ 195\ 200$ valores dos descritores.

Normalização dos descritores:

Devido ao método que será utilizado para fazer a classificação dos descritores, estes necessitam de ser normalizados para uma gama pertencente ao intervalo [0,1].

A normalização dos descritores é realizada num *kernel* de 380 blocos com 610 *threads* cada um, ou seja, um *kernel* que será lido 231800 vezes de forma paralela. Cada uma dessas *threads* vai analisar um descritor de 264 posições e dividir cada posição por 1358 caso se trate de um valor resultante de um histograma LBP (primeiras 256 posições) ou por 2256 caso se trate de um valor resultante de um histograma de gamas de cor (posições 257 a 264 do descritor). Estes valores para a normalização foram obtidos por análise do modelo de treino da SVM (secção 3.3), treino esse

realizado antes de todo o processo. Em casos em que a divisão resulte num valor superior a 1, o valor dessa posição do descritor tomará o valor de 1, de modo a não se ultrapassar o intervalo estabelecido.

O resultado da normalização será então 231 800 descritores com 264 posições cada um, que tomam valores na escala [0,1], e esta fase é necessária para uma correta classificação por parte da SVM.

Classificação dos descritores:

A classificação dos descritores é a última fase do processamento de imagem, quando, através da SVM previamente treinada se vai classificar cada um dos 231 800 descritores como sendo tronco, ou não.

Em primeiro lugar, foram procuradas bibliotecas *open source* de SVMs já existentes que tentem aplicar de forma paralela e otimizada uma correta classificação, como a GPULibSVM[6], a GTSVM[7], a GPUSVM [43], a ThunderSVM[8], entre outras. Mas todas as bibliotecas testadas não funcionaram, ora por serem apenas funcionais com versões de CUDA mais antigas, ora por funcionarem apenas com linguagens de programação específicas, como R ou Python, ou então devido a outros problemas de compatibilidade. Desta forma, chegou-se à conclusão que a melhor abordagem a tomar neste momento seria uma implementação manual das funções da libSVM que fossem necessárias, pois esta biblioteca já havido sido utilizada na implementação CPU do ViTruDe, como foi referido na secção 3.3.

Assim, foram programadas manualmente as funções da libSVM que permitissem uma classificação dos descritores *hLBP plus colour* em duas classes, "Tronco" ou "Não Tronco", numa SVM linear.

O algoritmo começa com a equação (3.2), em que, para um dado descritor, cada uma das suas 264 posições é multiplicada pela posição correspondente de cada um dos 128 SVs dados pelo modelo de treino. É de recordar que cada SV tem igualmente 264 posições. Desta equação resulta um vetor "Resultado" de 128 posições e cada uma delas armazena a multiplicação entre o descritor e um dos SVs.

$$\sum_{j=1}^{264} \text{Descritor}[j] \times \text{SV}_N[j] = \text{Resultado}[N] \ , \text{ para } N \in [1,128] \tag{3.2}$$

Cada uma das 128 posições deste vetor "Resultado" vai ser multiplicada pelo coeficiente respetivo (dado pelo modelo de treino), resultando um único valor total para o descritor, como é representado na equação (3.3).

$$\text{Total} = \sum_{N=1}^{128} \text{Coeficiente}[N] \times \text{Resultado}[N] \tag{3.3}$$

[6] http://mklab.iti.gr/project/GPU-LIBSVM
[7] http://ttic.uchicago.edu/ cotter/projects/gtsvm/
[8] https://thundersvm.readthedocs.io/en/latest/

A classificação será efetuada com recurso a probabilidades, mas antes disso, é aplicada a função sigmoide[9] como função de ativação da SVM, (3.4) e (3.5). A saída desta função é dada em intervalos de valores de [0,1], daí a necessidade prévia da normalização, e esta é uma função de ativação bastante utilizada em redes neuronais devido à sua facilidade de introdução de não-linearidades nos modelos [44].

$$\text{Soma} = \text{Total} \times (probA) + (-probB) \tag{3.4}$$

Os valores *probA* e *probB* da equação (3.4) são parâmetros resultantes do modelo de treino e representam as distribuições logísticas ajustadas aos valores de decisão dos classificadores binários.

$$\text{Final} = \begin{cases} \frac{e^{-Soma}}{1+e^{-Soma}}, & \text{Se Soma} > 0 \\ \frac{1}{1+e^{Soma}}, & \text{Se Soma} < 0 \end{cases} \tag{3.5}$$

Obtida a variável denominada de "Final" na equação (3.5), é aplicado um método de determinação de probabilidades de múltiplas classes [45], em que para poder ser efetuado, tem de ter como entrada duas probabilidades emparelhadas, calculadas a partir do resultado de "Final" com as equações (3.6) e (3.7).

$$\text{Primeira probabilidade emparelhada} = \min(\max(\text{Final}, 1x10^{-7}), 1 - 1x10^{-7}) \tag{3.6}$$

$$\text{Segunda probabilidade emparelhada} = 1 - \text{Primeira probabilidade emparelhada} \tag{3.7}$$

Após ser aplicado este método, resultam duas probabilidades distintas, a probabilidade do descritor pertencer à classe "Tronco" e a probabilidade do descritor pertencer à classe "Não Tronco". As duas são comparadas e daí conclui-se se se está perante um *pixel* correspondente a tronco ou não.

Esta é uma explicação elementar do funcionamento da classificação da libSVM para uma classificação de duas classes, e todos os procedimentos descritos são apenas para um descritor, portanto o método a realizar para os 231799 descritores restantes será em tudo semelhante.

Para realizar então a classificação dos descritores de forma concorrente, é lançado um *kernel* de 380 blocos com 610 *threads* cada, que permita efetuar 231800 execuções paralelas do *kernel*, de forma a se poder realizar estas operações em todos os descritores ao mesmo tempo. Foram passados como parâmetros os descritores normalizados, os SVs, os seus respetivos coeficientes e um vetor de 231800 posições, vazio, para guardar os valores da classificação de cada *pixel*. Este

[9] Função matemática caracterizada pela sua curva em S - curva sigmoide.

vetor, depois de preenchido no final da execução do *kernel*, é depois convertido novamente para CPU para ser tratado e possibilitar a visualização dos resultados em forma de imagem.

Toda esta computação torna esta fase de processamento a mais demorada e sobrecarregada. Cada *thread* tem que realizar várias operações e acessos a endereços de memória não adjacentes, o que se vai refletir nos tempos de processamento explorados no capítulo 4.

3.4.1 Aprimoramento do Algoritmo

Como se estudou na secção 2.4.3 do capítulo 2, os acessos devem ser feitos sempre de forma unida e alinhada, o que nem sempre é possível. Quando isto não acontece, o tempo de acesso às diversas variáveis por parte das *threads* aumenta consideravelmente. Este é um problema complexo e existem diversos estudos acerca da otimização dos acessos aos endereços de memória em CUDA [46]. Na Figura 3.8 está representado um exemplo de um acesso à memória não unido nem alinhado.

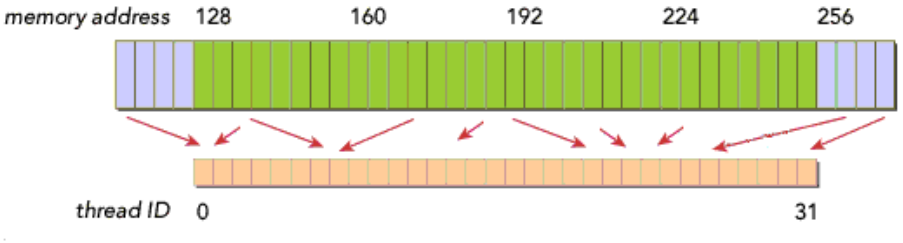

Figura 3.8: Exemplo de um mau acesso a endereços de memória. Figura adaptada de [6].

Numa fase final, após toda a implementação estar realizada, o algoritmo foi aprimorado de modo a se tentar obter tempos de processamento ainda menores.

As fases mais lentas do ViTruDe, nomeadamente o cálculo e a classificação dos descritores, são as fases computacionalmente mais pesadas e apresentam-se como sendo o *bottleneck* do algoritmo, e para diminuir os seus efeitos, foi implementado um processamento que ignora dois terços do número total de *pixels* processado nas fases de operações com os descritores (cálculo, normalização e classificação). De um modo global, o resultado final obtido, relativamente à deteção dos elementos naturais das vinhas propriamente dito, mantém-se como um resultado utilizável por parte do AgRob, e assim sendo, este mantém a ideia geral de onde se encontram os troncos e mastros, mas desta vez diminuindo consideravelmente o seu tempo de processamento. Os resultados desta abordagem são abordados em maior detalhe no capítulo 4.

Foi também realizada uma otimização no número de blocos e *threads* lançadas por *kernel*, tentando rentabilizar a taxa de ocupação das *warps* ao máximo, bem como diminuir o número de blocos utilizados. Por exemplo, quando se lançou 231800 *threads* na normalização dos descritores, esse número de *threads* não é múltiplo de 32, a dimensão das *warps*, e por isso foram lançados

227 blocos com 1024 *threads* cada, de maneira a reduzir o número de blocos e aumentar o número de *threads* para aumentar a taxa de ocupação dos núcleos da GPU, não deixando tantos recursos inutilizados ou com uma utilização parcial. O número máximo de *threads* por bloco é 1024, tal como pode ser observado no anexo B.

Concluindo, o pretendido é que a execução das tarefas do AgRob V16 seja realizada, o mais possível, em tempo real, e assim estas otimizações no algoritmo foram implementadas nesse sentido. De seguida, no capítulo 4, são apresentados os resultados obtidos para a implementação descrita ao longo deste capítulo 3.

Capítulo 4

Testes e Resultados

Neste capítulo são demonstrados os testes efetuados e os resultados relevantes a destacar.

A secção 4.1 mostra os resultados da abordagem GPU do ViTruDe relativamente à sua deteção de elementos naturais das vinhas propriamente dita. Para averiguar se os métodos adotados para acelerar este processo degradaram os resultados relativamente ao processamento original, é também feita uma comparação da precisão deste processamento visual com o que havia sido feito em CPU.

Já a secção 4.2 mostra o tempo de processamento das diferentes fases do ViTruDe, comparando os resultados obtidos para o processamento original em CPU e para a abordagem desta dissertação, em GPU.

Os testes foram realizados em Ubuntu 16.04 LTS (64-bit) e processador (CPU) Intel® Core™ i7-4700HQ, para dois diferentes GPUS: NVIDIA GeForce GT 740M e NVIDIA GeForce GTX 1050 TI. As *datasets* reais com as *frames* analisadas, armazenadas em *bags*[1], foram obtidas via ROS[2].

4.1 Resultados do ViTruDe em GPU

Como havia sido referido previamente na secção 3.3, é esperada uma boa deteção de troncos para condições semelhantes ao *dataset* utilizado para treino e teste da SVM, nomeadamente para *frames* retiradas em condições meteorológicas e climatéricas similares e com definições de câmara idênticas, ou seja, *frames* retiradas nas videiras da UTAD numa manhã de Sol com uma câmara sem infravermelhos nem filtro azul. Toda a informação acerca das condições das *frames* utilizadas está publicada online e com livre acesso[3].

Na Figura 4.1 pode observar-se exatamente uma imagem em condições idênticas às do *dataset* utilizado para o treino e teste da SVM, bem como o resultado final obtido pelo processamento do ViTruDe em GPU. A branco estão representados os *pixels* nos quais é provável a existência de troncos.

[1] Ficheiros ROS que são responsáveis pelo armazenamento de dados e que permitem a sua posterior reprodução.
[2] Informação mais detalhada em: http://wiki.ros.org/rosbag
[3] http://vcriis01.inesctec.pt

Figura 4.1: À esquerda, uma imagem captada pelo AgRob nas videiras da UTAD, de manhã, com uma câmara sem infravermelhos nem filtro azul, e o respetivo resultado obtido pelo processamento do ViTruDe em GPU, à direita.

Para este género de imagens, para as quais a SVM foi treinada e testada, verifica-se que, de um modo global, a deteção de troncos nas vinhas teve sucesso. Verifica-se também a existência de diversos falsos positivos e falsos negativos, os quais podem ser reduzidos com um treino usando amostras em quantidade superior, o mais similares possíveis com o cenário real de teste. Mais especificamente, será de interesse utilizar os falsos positivos obtidos nos testes para futuras amostras de treino da classe "Não Tronco".

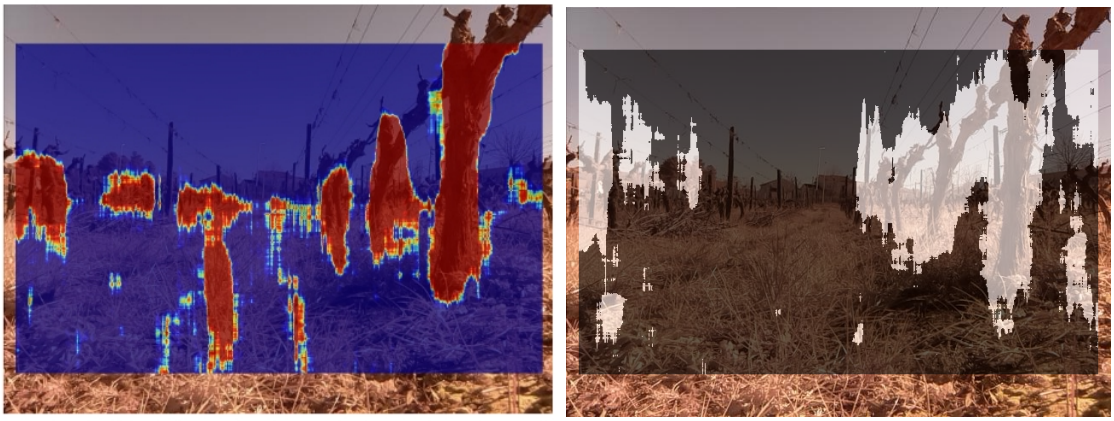

Figura 4.2: Comparação dos resultados do ViTruDe em CPU (esquerda) e GPU (direita) para a mesma imagem de entrada.

Na Figura 4.2 é feita a comparação de resultados da implementação do ViTruDe em CPU e para GPU. A imagem utilizada para estas duas execuções é a mesma imagem de entrada da Figura 4.1, portanto, trata-se de uma imagem captada nas videiras da UTAD numa manhã de Sol com uma câmara sem infravermelhos nem filtro azul. Esta comparação é feita, em ambas as implementações, para a configuração do ViTruDe, em que é utilizado um descritor do tipo *plus colour* e SVM de núcleo linear com duas classes. A vermelho, na ilustração da implementação

em CPU, estão os *pixels* em que é provável a existência de troncos, enquanto que, na ilustração em GPU, esses *pixels* estão a branco, como já foi observado acima.

Comparando os dois resultados, pode-se constatar que, apesar de ambos apresentarem falsos positivos, a deteção de elementos naturais nas vinhas é realizada satisfatoriamente, e com um treino adequado, a abordagem do ViTruDe em GPU apresenta resultados bastante fiáveis, especialmente para troncos localizados a curtas e médias distâncias.

A Figura 4.3 apresenta mais um exemplo da execução do ViTruDe, em GPU.

Figura 4.3: À esquerda, uma imagem captada pelo AgRob nas videiras da UTAD, de tarde, com uma câmara sem infravermelhos nem filtro azul, e o respetivo resultado obtido pelo processamento do ViTruDe em GPU, à direita.

Esta figura representa um cenário idêntico à situação da Figura 4.1, exceto que desta vez esta *frame* foi observada durante a tarde e não de manhã.

Figura 4.4: À esquerda, uma imagem captada pelo AgRob nas videiras da UTAD, de tarde, com uma câmara RGB, e o respetivo resultado obtido pelo processamento do ViTruDe em GPU, à direita.

Como forma de comprovar que o treino da SVM para as características desejadas é importantíssimo, foi incluída nesta secção a Figura 4.4, representativa dos resultados para uma *frame*

num cenário não treinado (o modelo de treino utilizado é o mesmo do que o utilizado para as imagens acima). Aqui, foi utilizada uma câmara RGB e pode-se aferir que os resultados não são os melhores, especialmente quando comparados com os resultados ilustrados nas Figuras 4.3 e 4.1.

Desta forma, é oportuno concluir que para se obter bons resultados alicerçados no ViTruDe, tem de haver um treino minucioso realizado à priori para o cenário das vinhas no qual é pretendido que o AgRob atue.

Como foi explicado mais detalhadamente na secção 3.4.1, para se conseguir reduzir ainda mais o tempo de processamento do algoritmo e para tirar proveito de se ter implementado manualmente a SVM, foi testada uma abordagem em que o algoritmo apenas processa um *pixel* em cada três, resultando num processamento de apenas um terço dos *pixels* totais. O resultado deste procedimento está representado na Figura 4.5.

Figura 4.5: Resultados de processar apenas um terço dos *pixels* da imagem.

Tal como anteriormente, as áreas a branco representam *pixels* onde é mais provável que se encontrem os troncos ou mastros. Com esta abordagem, o AgRob continua a obter com sucesso a localização dos elementos naturais das vinhas, fazendo-o de forma mais eficiente.

Concluindo, a implementação em GPU do ViTruDe mantém a boa precisão do processamento visual da sua abordagem em CPU. Todavia, bons resultados a este nível já haviam sido atingidos. O que está realmente em estudo nesta dissertação é se uma implementação em GPU é viabilizada por tornar este processamento mais rápido, ou não. Esta análise será feita na secção seguinte, 4.2.

4.2 Tempo de Processamento

A vantagem da implementação ter sido realizada de uma forma faseada é que permitiu ir verificando os tempos que demoram os diferentes segmentos do processamento de imagem do ViTruDe. Isto permitiu não só verificar o tempo global total de todo o processo, como também verificar onde se encontram as fases mais lentas e prioritárias no sentido de serem alvo de futuras melhorias.

Porque os resultados são largamente influenciados pelo *hardware* e condições praticadas, mencionar os processadores CPU e GPU utilizados nos testes é bastante relevante. Os testes foram realizados em ambiente de trabalho Ubuntu 16.04 LTS e os processadores para cada abordagem encontram-se na Tabela 4.1.

Tabela 4.1: Especificações do *hardware* para os testes realizados

CPU	Intel® Quad Core™ i7-4700HQ
GPU1	NVIDIA GeForce GT 740M
GPU2	NVIDIA GeForce GTX 1050 TI

Algumas conclusões interessantes podem ser adquiridas analisando inicialmente a primeira fase do processamento que sofreu intervenções diretas de uma programação heterogénea: a obtenção da imagem LBP. Na Tabela 4.2 pode observar-se os diferentes tempos de processamento obtidos, em segundos, com implementações em CPU, GPU de forma sequencial e GPU de forma paralelizada para a saída direta da imagem de entrada (ler todos os *pixels* e formar imagem de saída que é igual à da entrada) e para extração da imagem LBP.

Tabela 4.2: Tempos obtidos para extração de uma imagem LBP para diferentes configurações de CPU e GPU

Implementação	**Tempos de processamento (segundos)**	
	Saída direta da imagem de entrada	Extração de imagem LBP
CPU	0,014552	0,017969
GPU1 (sequencial)	0,070132	0,151646
GPU1 (paralelizado)	0,000469	0,000452

O teste para a GPU1 (GeForce GT 740M) de forma sequencial foi apenas realizado como confirmação de que o CPU, mesmo assim, é bastante mais rápido do que o GPU quando este atua de forma sequencial, em tudo semelhante à execução em CPU. Verifica-se que execuções em CPU são mais rápidas do que execuções sequenciais em GPU, mal otimizadas e que não tiram partido da sua capacidade de paralelização.

Pode-se constatar então que a implementação do processamento de imagem em GPU de forma paralelizada é muito mais rápida do que a sua implementação em CPU, apresentando tempos de processamento para uma extração de imagem LBP de, neste caso, cerca de 40 vezes menores, o que se apresenta como sendo uma melhoria significativa. A implementação em GPU, mas de

forma sequencial, apresenta os piores resultados a nível de tempo de processamento. Esta implementação utiliza apenas uma única *thread* para processar toda a imagem, o que naturalmente atrasa todo o processo. Isto vai ao encontro do que havia sido mencionado no capítulo 2.4, ou seja, uma implementação em GPU apenas se justifica quando existe uma enorme quantidade de dados a serem processados e estes apresentam elevada possibilidade de correrem de forma concorrente. Se não for esse o caso, a melhor escolha recai sobre o CPU devido à sua capacidade de processar lógica complexa (tal como representado na Figura 2.17 do capítulo 2).

Os resultados para o ViTruDe completo estão ilustrados na Tabela 4.3 e foram obtidos fazendo a média aritmética dos tempos para vinte execuções da mesma *frame* de entrada - a imagem da Figura 4.1. Nesta tabela estão os resultados para os processamentos completos de todos os *pixels* da imagem, assim como uma abordagem extra, detalhada na secção 3.4.1, que faz o processamento da imagem saltando de três em três *pixels* nas secções do ViTruDe de cálculo, normalização e classificação dos descritores, ou seja, processa apenas um terço da imagem total nessas fases.

Tabela 4.3: Comparação entre os tempos de processamento do ViTruDe em CPU e GPU

Funções	**Tempo de Processamento (segundos)**				
	Todos os *Pixels*			Saltando Três *Pixels*	
	CPU	GPU1	GPU2	GPU1	GPU2
Extração LBP	0,017969	0,000452	0,000193	0,000491	0,000194
Extração HSV	0,008083	-	-	-	-
Cálculo dos Descritores	≈ 600	7,383115	0,976206	3,633317	0,471755
Normalização dos Descritores		0,696364	0,174869	0,090228	0,018046
Classificação dos Descritores		34,709829	6,238318	5,998721	0,890845
Tempo Total	≈ 10 minutos	42,80 segundos	7,40 segundos	9,73 segundos	1,39 segundos

A tabela mostra que, para um processamento completo, uma abordagem em GPU é mais eficiente a nível de tempo de processamento. Nos testes efetuados, com as especificações de *hardware* mencionadas, a implementação em GPU do ViTruDe no GPU1 é cerca de 14 vezes mais rápida quando comparada com o mesmo procedimento em CPU, enquanto que a mesma abordagem no GPU2 é mais de 80 vezes mais rápida do que o ViTruDe em CPU e cerca de seis vezes mais rápida do que o GPU1. Olhando especificamente, mais uma vez, para a extração LBP, a diferença é ainda maior quando se utiliza o GPU2: 93 vezes mais rápido do que o CPU.

Como foi explicado no capítulo 3, tira-se proveito da execução concorrente dos *kernels* CUDA relativamente ao código *host* para extrair a imagem HSV em CPU paralelamente à extração da imagem LBP em GPU. É devido a isto que a extração da imagem HSV na tabela não apresenta nenhum valor para as colunas das abordagens em GPU. Então, o tempo total conjunto da extração das imagens LBP e HSV será o mais lento dos dois, devido à paralelização destes dois processos.

Concluindo, o processamento do ViTruDe, quando implementado utilizando o modelo de programação CUDA, tirando proveito da capacidade de paralelização em GPU, leva apenas $1,2\%$ do tempo de execução que o mesmo procedimento leva em CPU.

Dito isso, devido a razões relacionadas com a estrutura geral do algoritmo, o cálculo e a classificação dos descritores demonstram ser o *bottleneck* do processo. Assim, para alcançar um processamento ainda mais eficiente e para tirar proveito de se ter implementado a SVM de forma manual, foi realizada uma abordagem em que o algoritmo apenas processa um terço de todos os *pixels* da imagem nas fases de operações com os descritores, sendo obtida a imagem já demonstrada acima, na Figura 4.5.

Com esta implementação, o AgRob faz a deteção dos elementos naturais das vinhas muito mais eficientemente: Aproximadamente 9 segundos para a NVIDIA GeForce GT 740M e cerca de 1 segundo para a NVIDIA GeForce GTX 1050 TI. Estes resultados são, naturalmente, muito mais rápidos relativamente aos resultados em GPU obtidos anteriormente, cerca de quatro vezes mais rápido para o GPU1 e cinco vezes para o GPU2. Estes resultados, em relação à implementação CPU, numa comparação um pouco injusta, visto que se está a comparar um processamento do número total dos *pixels* em CPU com um processamento parcial dos mesmos em GPU, demonstram, como seria de esperar, uma melhoria muito significativa, mais de 60 vezes mais rápido no GPU1 e acima de 400 vezes para o GPU2.

Esta abordagem, especialmente no GPU2, aproxima o ViTruDe da velocidade de execução pretendida para uma aplicação em tempo real, sem comprometer demasiado a qualidade dos resultados obtidos a nível da perceção visual dos elementos naturais das vinhas.

Capítulo 5

Conclusões e Trabalho Futuro

Este capítulo está dividido em duas secções e tem a finalidade de sintetizar as tarefas que foram executadas, bem como exibir algumas considerações acerca dos resultados obtidos.

Na secção 5.1 são apresentadas as principais conclusões obtidas referentes ao trabalho desenvolvido durante esta dissertação e averigua-se se os objetivos propostos para os trabalhos do semestre foram atingidos.

Na secção 5.2 são sugeridas algumas propostas de trabalho complementar que poderia ser realizado no futuro com vista à melhoria e otimização dos módulos desenvolvidos.

Foram utilizados repositórios de código[1] e de documentos[2], pertencentes ao INESC TEC.

5.1 Satisfação dos Objetivos

O objetivo desta dissertação consistia em tornar o processamento de imagem do AgRob, responsável pela deteção de troncos de videiras como meio de auxílio à sua localização, mais rápido, de maneira a tornar os trabalhos nas vinhas o mais eficientes possível.

Para tal decidiu-se reutilizar o algoritmo ViTruDe, mas tentando provocar a sua aceleração através de métodos de paralelização por programação em GPU. Foi feito o estudo prévio acerca das particularidades do algoritmo e da plataforma de computação paralela CUDA, desenvolvida pela NVIDIA como modelo de programação para as suas placas gráficas, como é o caso da plataforma robótica em questão.

A programação foi feita de forma faseada, sendo cada fase do processamento implementada e testada cuidadosamente, tendo-se conseguido obter resultados bastante satisfatórios, demonstrados em detalhe no capítulo 4.

Relativamente ao processador da plataforma robótica, NVIDIA Tegra K1, teria sido igualmente interessante ver o seu desempenho com o algoritmo desenvolvido, porém, isso não foi possível porque, primeiro, a NVIDIA já deixou de suportar esse processador e a sua última versão

[1] Repositório de código pode ser acedido através de: https://gitlab.inesctec.pt/

[2] Repositório de documentos pode ser acedido através de: http://criis-projects.inesctec.pt/

CUDA disponível possui imensas incompatibilidades com a versão CUDA para o qual os trabalhos foram integralmente realizados (versão 6.5 ao invés da 9.1 utilizada nesta dissertação). Além disso, a equipa que trabalha no projeto ROMOVI tem planos para substituir este processador por outro mais potente num futuro próximo (também da NVIDIA, de maneira que será possível fazer uso dos trabalhos desenvolvidos nesta dissertação), e assim sendo, trabalhar num processador desatualizado e que vai ser substituído em pouco tempo tornou-se desnecessário para as conclusões a retirar deste trabalho. Contudo, todas as suas propriedades e especificações são apresentadas no anexo B para comparação com as GPUs testadas.

Atendendo ao facto de que se estava a otimizar um algoritmo que demorava cerca de 10 minutos a executar em CPU, houve melhorias significativas no que ao tempo de execução diz respeito. Em média, para o GPU mais poderoso que foi testado, NVIDIA GeForce GTX 1050 TI, o algoritmo ViTruDe demora cerca de 7 segundos e meio a executar para exatamente as mesmas condições do CPU, um valor mais de 80 vezes inferior ao da sua implementação original. Noutras palavras, a execução do ViTruDe em GPU demora apenas $1,2\%$ do seu tempo de execução em CPU.

Concluindo, o objetivo principal da dissertação foi assim cumprido. O ViTruDe é executado de uma forma muito mais célere, porém, ainda se pode tornar mais rápido e para isso serão apresentadas algumas propostas de trabalho futuro que vão neste sentido.

5.2 Trabalho Futuro

De seguida, são sugeridos um conjunto de pontos que poderão ser explorados no futuro de modo a melhorar os resultados obtidos do algoritmo ViTruDe em GPU em relação ao seu tempo de processamento.

Testar Implementação em OpenACC:

O OpenACC[3] é um modelo de programação complementar ao CUDA que usa uma API baseada em diretivas e projetada para alta *performance*, de fácil programação e portabilidade entre plataformas [47]. As diretivas são usadas para comunicar ao compilador a forma como os diferentes trechos de códigos devem ser paralelizados.

O modelo de *threads* do OpenACC é similar ao do CUDA e o seu paralelismo é dividido em gangues, *workers* e vetores. No nível superior, os gangues são análogos aos blocos do CUDA. Cada gangue é submetido a um processador e possui uma ou mais *threads* de execução. Dentro de cada gangue existe um ou mais *workers*, análogos em CUDA a *warps*. Cada *worker* tem um vetor de uma determinada dimensão, consistindo num ou mais elementos desse vetor que executam a mesma instrução ao mesmo tempo (em CUDA isto é análogo à dimensão do *warp*, e cada elemento do vetor representa uma *thread*).

A principal diferença entre o OpenACC e o CUDA é que em OpenACC é exposto ao utilizador

[3] https://www.openacc.org/

o conceito de *workers* diretamente no modelo de programação, enquanto que em CUDA não se cria explicitamente *warps* de *threads*. Em OpenACC a terminologia também é diferente daquela usada em CUDA.

O OpenACC oferece assim mais controlo de execução relativamente ao CUDA e requer menor esforço manual de gestão comunicacional, eliminando muitas das tarefas mundanas da programação GPU, como a transferência de dados entre o *host* e o *device*. É por isso mais fácil de utilizar, intuitivo e flexível do que o modelo CUDA, contudo, a sua *performance* é geralmente menor quando comparada com implementações cuidadas e manuais de código em CUDA [48].

Apesar de serem modelos diferentes, o CUDA e o OpenACC podem ser usados em conjunto para o desenvolvimento mais rápido e eficaz de aplicações paralelizadas [6]. Relativamente ao ViTruDe em concreto, a flexibilidade do OpenACC poderá oferecer um bom complemento ao código desenvolvido em CUDA, sendo o seu uso conjunto um bom exercício de comparação e teste para ser eventualmente desenvolvido no futuro.

Alterações ao Algoritmo:

Como foi comprovado durante a implementação e teste do algoritmo, os métodos de processamento de imagem utilizados, nomeadamente a utilização de descritores em forma de histograma e as tarefas que envolvem a sua classificação pela SVM, são bastante complexos e oferecem grandes dificuldades para uma implementação de acessos unidos aos endereços de memória por parte das *threads* (ver secção 4.3.1 do capítulo 4), contribuindo para um aumento do tempo de processamento.

Um trabalho futuro poderia passar pela alteração do ViTruDe, estudando a utilização de outros métodos de filtragem expedita de descritores não relevantes, otimizados para aplicação em programação paralela. Deste modo, iriam permitir uma paralelização mais rápida, obrigando as *threads* a uma sobrecarga de tarefas menor, tornando o algoritmo mais eficiente.

Utilização de biblioteca *open source* SVM devidamente otimizada para utilização em GPU:

As fases de processamento que demoraram mais tempo a executar foram aquelas que envolvem as operações com os descritores. Como foi clarificado na secção 4.3 do capítulo 4, as funções da libSVM necessárias para a classificação foram implementadas manualmente em GPU, isto porque depois de alguma pesquisa, não foi encontrada nenhuma biblioteca *open source* capaz de as realizar eficientemente, para além de ser mais flexível fazer essas funções manualmente.

Contudo, isto não significa que no futuro não surjam boas bibliotecas *open source* de uma SVM, devidamente otimizadas para utilização em GPU, e que garantam tempos de processamento melhores do que os alcançados. Esta é assim uma sugestão que poderá vir a ser implementada de modo a tornar as tarefas do AgRob V16 mais breves, de forma o mais possível em tempo real.

Anexo A

Configurações

Esta dissertação foi desenvolvida em sistema operativo Linux, distribuidor Ubuntu 16.04 LTS e necessitou de várias configurações, nomeadamente a configuração do ROS Kinetic para utilização com OpenCV (versão 3.3.1) em conjunto com o CUDA (versão 9.1).

De forma a servir de auxílio a futuros trabalhos, abaixo encontra-se o ficheiro CMakeLists.txt[1] do projeto em ROS, vital para a sua compilação nas configurações referidas acima. Relembra-se que este anexo apenas demonstra o que foi realizado no CMake e não retrata qualquer código desenvolvido relativamente à programação em CPU propriamente dita.

A azul estão representados comentários, enquanto que o código transcrito a verde representa o que é modificável, como os nomes dos ficheiros, por exemplo. A vermelho está a capacidade CUDA da placa gráfica, que varia consoante a GPU utilizada - Anexo B.

A.1 CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(Project Name)

## Find Catkin Macros And Libraries

find_package(catkin REQUIRED COMPONENTS
        roscpp
        rospy
        std_msgs
        image_transport
        image_geometry
        cv_bridge
        rosbag
        geometry_msgs
```

[1] http://wiki.ros.org/catkin/CMakeLists.txt

```
        tf)

## System Dependencies Are Found With CMake's Conventions

find_package(OpenCV REQUIRED)
include_directories(${OpenCV_INCLUDE_DIRS})
find_package(Boost REQUIRED COMPONENTS system thread filesystem)
include_directories(${BOOST_INCLUDE_DIR})
find_package(CUDA REQUIRED)
include_directories(
  ${CATKIN_INCLUDE_DIRS}
  ${CUDA_INCLUDE_DIRS})

## Catkin Specific Configuration

catkin_package(
)

## Build Configuration

set (CUDA_NVCC_FLAGS "-arch=sm_Compute Capability "
        CACHE STRING "nvcc flags" FORCE)
set (CUDA_VERBOSE_BUILD ON CACHE BOOL "nvcc verbose" FORCE)
set (LIB_TYPE STATIC)
CUDA_add_library( Library Name  ${LIB_TYPE} src/cuda_code_file.cu)

set (CUDA_NVCC_FLAGS "${CUDA_NVCC_FLAGS} -DMY_DEF=1")
set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -DMY_DEF=1")
set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -DMY_DEF=1")

link_directories(${CUDA_LIBRARY_DIRS})
cuda_compile( Executable File  src/cuda_code_file.cu)

## Add Cmake Target Dependencies Of The Library

set(FBN_SVM       src/fbn_libsvm/fbn_svm.cpp
                  src/lrf_to_camera.cpp
                  src/fbn_libsvm/fbn_svm-predict.cpp
                  src/fbn_libsvm/fbn_svm-scale.cpp
                  src/fbn_libsvm/fbn_svm-train.cpp)
```

```
set(FBN_LBP       src/fbn_lbp/lbp_pattern.cpp
                  src/fbn_lbp/imagedescriptor.cpp)

## Declare C++ And CUDA Executable

add_executable(C++ Executable Name   src/main.cpp
               ${FBN_SVM})
target_link_libraries( C++ Executable Name
                       ${catkin_LIBRARIES}
                       ${OpenCV_LIBS}
                       ${cuda_LIBRARIES})

add_executable(CUDA Executable Name src/cpp_code_file.cpp  src/train_and_test.cpp
               ${FBN_SVM}
               ${FBN_LBP}
               ${Executable File})
target_link_libraries(CUDA Executable Name
                       ${catkin_LIBRARIES}
                       ${OpenCV_LIBS}
                       ${cuda_LIBRARIES}
              Library Name   Path To .hpp Header File )
```

Anexo B

Tabela de Propriedades CUDA

Na Tabela B.1 podem ser observadas e comparadas as propriedades de três GPUs para programação paralela: GeForce GT 740M, Tegra K1 e GeForce GTX 1050 TI, todas da NVIDIA.

Tabela B.1: Comparação entre propriedades CUDA entre GeForce GT 740M, Tegra K1 e GeForce GTX 1050 TI

Propriedades CUDA	**GeForce GT 740M**	**Tegra K1**	**GeForce GTX 1050 TI**
Capacidade CUDA	3.0	3.2	6.1
Nº de Multiprocessadores	2	1	6
Núcleos CUDA por Multiprocessador	192	192	128
Nº Total de Núcleos CUDA	384	192	768
Memória Global Total	2048 MBytes	2048 MBytes	4096 MBytes
Frequência do *Clock* do GPU	895 MHz	852 MHz	1620 MHz
Frequência do *Clock* de Memória	800 MHz	924 MHz	3504 Mhz
Largura do Barramento de Memória	128-bit	64-bit	128-bit
Tamanho da Memória *Cache* L2	262144 bytes	131072 bytes	1048576 bytes
Dimensões de Textura (x,y,z)	1D = (65536) 2D = (65536,65536) 3D = (4096,4096,4096)	1D = (65536) 2D = (65536,65536) 3D = (4096,4096,4096)	1D = (131072) 2D = (131072, 65536) 3D = (16384,16384,16384)
Textura em Camadas (dimensão) × camadas	1D = (16384)×2048 2D = (16384,16384)×2048	1D = (16384)×2048 2D = (16384,16384)×2048	1D = (32768)×2048 2D = (32768,32768)×2048
Memória Constante Total	65536 bytes	65536 bytes	65536 bytes
Memória Partilhada por Bloco	49152 bytes	49152 bytes	49152 bytes
Registos Disponíveis por Bloco	65536	32768	65536
Tamanho do *Warp*	32	32	32
Máx. nº de *Threads* por Multiprocessador	2048	2048	2048
Máx. nº de *Threads* por Bloco	1024	1024	1024
Tamanhos Máx. por Dimensão de Bloco	1024 × 1024 × 64	1024 × 1024 × 64	1024 × 1024 × 64
Tamanhos Máx. por Dimensão de Grelha	2147483647 × 65356 × 65356	2147483647 × 65356 × 65356	2147483647 × 65356 × 65356
Pitch de Memória Máximo	2147483647 bytes	2147483647 bytes	2147483647 bytes

Publicações Relacionadas

Durante a realização desta dissertação, foi desenvolvido e submetido o seguinte artigo:

«Parallelization of a Vine Trunk Detection Algorithm For a Real Time Robot Localization System»

Filipe Azevedo, Pranjali Shinde, Luis Santos, Jorge Mendes, Filipe Neves dos Santos e Hélio Mendonça.

Journal: *Computers and Electronics in Agriculture*

Publisher: *Elsevier*

«Computers and Electronics in Agriculture provides international coverage of advances in the development and application of computer hardware, software and electronic instrumentation and control systems for solving problems in agriculture and related industries.»

ISSN: 0168-1699

Referências

[1] Czesław Puchalski, Józef Gorzelany, Grzegorz Zaguła, e Gerald Brusewitz. Image Analysis For Apple Defect Detection. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.501.781&rep=rep1&type=pdf.

[2] M. Guijarro, G. Pajares, I. Riomoros, P.J. Herrera, X.P. Burgos-Artizzu, e A. Ribeiro. Automatic segmentation of relevant textures in agricultural images. *Computers and Electronics in Agriculture*, 75(1):75–83, 1 2011. URL: https://www.sciencedirect.com/science/article/pii/S0168169910001924, doi:10.1016/J.COMPAG.2010.09.013.

[3] Andrew English, Ross, Patrick, David Corke, e Peter. Vision Based Guidance for Robot Navigation in Agriculture. páginas 1693–1698, 2014. URL: https://doi.org/10.1109/ICRA.2014.6907079, doi:10.1109/ICRA.2014.6907079.

[4] Página Web do AgRob, 2018. URL: http://www.agrob.inesctec.pt.

[5] Jorge Mendes, Filipe Neves Dos Santos, Nuno Ferraz, Pedro Couto, e Raul Morais. Vine Trunk Detector for a Reliable Robot Localization System. Em *Proceedings - 2016 International Conference on Autonomous Robot Systems and Competitions, ICARSC 2016*, 2016. doi:10.1109/ICARSC.2016.68.

[6] John Cheng, Max Grossman, e Ty McKercher. *Professional CUDA C Programming*, volume 53. 2013. doi:10.1017/CBO9781107415324.004.

[7] Lalit Saxena, Leisa Armstrong, Lalit P Saxena, e Leisa J Armstrong. A survey of image processing techniques for agriculture. URL: http://ro.ecu.edu.au/ecuworkspost2013/854.

[8] Yan Ma, Lajiao Chen, Peng Liu, e Ke Lu. Parallel programing templates for remote sensing image processing on GPU architectures: design and implementation. *Computing*, 98(1-2):7–33, 2016. doi:10.1007/s00607-014-0392-y.

[9] CRIIS. Página web do ROMOVI, 2017. URL: http://criis.inesctec.pt/index.php/criis-projects/romovi/.

[10] C. Wouter Bac, Eldert J. van Henten, Jochen Hemming, e Yael Edan. Harvesting Robots for High-value Crops: State-of-the-art Review and Challenges Ahead. *Journal of Field Robotics*, 31(6):888–911, 11 2014. URL: http://doi.wiley.com/10.1002/rob.21525, doi:10.1002/rob.21525.

[11] Rainer Bischoff e KUKA Roboter Tim Guhl. The strategic research agenda for robotics in Europe. *Technology*, (March):15–16, 2009.

[12] João Rebelo e Leonida Correia. Port wine dynamics: Production, trade and market structure. *Regional and Sectoral Economic Studies*, 8(1):113–114, 2008.

[13] António C. Real, José Borges, José S. Cabral, e Gregory V. Jones. A climatology of Vintage Port quality. *International Journal of Climatology*, 37(10):3798–3809, 8 2017. URL: http://doi.wiley.com/10.1002/joc.4953, doi:10.1002/joc.4953.

[14] David J. Mulla. Twenty five years of remote sensing in precision agriculture: Key advances and remaining knowledge gaps. *Biosystems Engineering*, 114(4):358–371, 4 2013. URL: https://www.sciencedirect.com/science/article/pii/S1537511012001419, doi:10.1016/J.BIOSYSTEMSENG.2012.08.009.

[15] Benoit A. Aubert, Andreas Schroeder, e Jonathan Grimaudo. IT as enabler of sustainable farming: An empirical analysis of farmers' adoption decision of precision agriculture technology. *Decision Support Systems*, 54(1):510–520, 12 2012. URL: https://www.sciencedirect.com/science/article/pii/S0167923612001972, doi:10.1016/J.DSS.2012.07.002.

[16] C.-C Yang, S O Prasher, J.-A Landry, J Perret, H S Ramaswamy, e J Ramaswamy. Recognition of weeds with image processing and their use with fuzzy logic for precision farming. *CANADIAN AGRICULTURAL ENGINEERING*, 42(4), 2000. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.5394&rep=rep1&type=pdf.

[17] Herman Eerens, Dominique Haesen, Felix Rembold, Ferdinando Urbano, Carolien Tote, e Lieven Bydekerke. Image time series processing for agriculture monitoring. *Environmental Modelling & Software*, 53:154–162, 3 2014. URL: https://www.sciencedirect.com/science/article/pii/S1364815213002648, doi:10.1016/J.ENVSOFT.2013.10.021.

[18] D S Jayas, J Paliwal, e N S Visen. Multi-layer Neural Networks for Image Analysis of Agricultural Products. *J. agric. Engng Res*, 77(2):119–128, 2000. URL: http://www.idealibrary.com, doi:10.1006.

[19] Yud-Ren Chen, Kuanglin Chao, e Moon S Kim. Machine vision technology for agricultural applications. *Computers and Electronics in Agriculture*, 36(2-3):173–191, 11 2002. URL: https://www.sciencedirect.com/science/article/pii/S016816990200100X, doi:10.1016/S0168-1699(02)00100-X.

[20] M. Montalvo, J.M. Guerrero, J. Romeo, L. Emmi, M. Guijarro, e G. Pajares. Automatic expert system for weeds/crops identification in images from maize fields. *Expert Systems with Applications*, 40(1):75–82, 1 2013. URL: https://www.sciencedirect.com/science/article/pii/S0957417412008895, doi:10.1016/J.ESWA.2012.07.034.

[21] Peilin Li, Sang-heon Lee, e Hung-Yao Hsu. Study on citrus fruit image data separability by segmentation methods. *Procedia Engineering*, 23:408–416, 1 2011. URL: https://www.sciencedirect.com/science/article/pii/S1877705811053653, doi:10.1016/J.PROENG.2011.11.2522.

[22] Martin Weis e Roland Gerhards. Weis, Gerhards Detection of weeds using image processing and clustering. URL: http://www2.atb-potsdam.de/CIGR-ImageAnalysis/images/16_115_Weis_Poster%20_2_.pdf.

[23] Minghua Zhang, Zhihao Qin, Xue Liu, e Susan L Ustin. Detection of stress in tomatoes induced by late blight disease in California, USA, using hyperspectral remote sensing. *International Journal of Applied Earth Observation and Geoinformation*, 4(4):295–310, 11 2003. URL: https://www.sciencedirect.com/science/article/pii/S0303243403000084, doi:10.1016/S0303-2434(03)00008-4.

[24] Xavier P. Burgos-Artizzu, Angela Ribeiro, Alberto Tellaeche, Gonzalo Pajares, e Cesar Fernández-Quintanilla. Analysis of natural images processing for the extraction of agricultural elements. *Image and Vision Computing*, 28(1):138–149, 1 2010. URL: https://www.sciencedirect.com/science/article/pii/S0262885609001140, doi:10.1016/J.IMAVIS.2009.05.009.

[25] Eliyahu (Efim) Kelman e Raphael Linker. Vision-based localisation of mature apples in tree images using convexity. *Biosystems Engineering*, 118:174–185, 2 2014. URL: https://www.sciencedirect.com/science/article/pii/S1537511013001918, doi:10.1016/J.BIOSYSTEMSENG.2013.11.007.

[26] J. Reid e S. Searcy. Vision-based guidance of an agriculture tractor. *IEEE Control Systems Magazine*, 7(2):39–43, 4 1987. URL: http://ieeexplore.ieee.org/document/1105271/, doi:10.1109/MCS.1987.1105271.

[27] Santosh Hiremath, Frits Van Evert, Gerie Van Der Heijden, Cajo Ter Braak, e Alfred Stein. Image-Based Particle Filtering For Robot Navigation In A Maize Field. URL: http://www.cs.cmu.edu/~mbergerm/agrobotics2012/02Hiremath.pdf.

[28] H.T. Søgaard e H.J. Olsen. Determination of crop rows by image analysis without segmentation. *Computers and Electronics in Agriculture*, 38(2):141–158, 2 2003. URL: https://www.sciencedirect.com/science/article/pii/S0168169902001400, doi:10.1016/S0168-1699(02)00140-0.

[29] Filipe Neves Dos Santos, Heber Sobreira, Daniel Campos, Raul Morais, António Paulo Moreira, e Olga Contente. Towards a reliable monitoring robot for mountain vineyards. Em *Proceedings - 2015 IEEE International Conference on Autonomous Robot Systems and Competitions, ICARSC 2015*, 2015. doi:10.1109/ICARSC.2015.21.

[30] Filipe Neves dos Santos, Heber Sobreira, Daniel Campos, Raul Morais, António Paulo Moreira, e Olga Contente. Towards a Reliable Robot for Steep Slope Vineyards Monitoring. *Journal of Intelligent and Robotic Systems: Theory and Applications*, 2016. doi:10.1007/s10846-016-0340-5.

[31] Marcos Duarte, Filipe Neves dos Santos, Armando Sousa, e Raul Morais. Agricultural Wireless Sensor Mapping for Robot Localization. páginas 359–370. Springer, Cham, 2016. URL: http://link.springer.com/10.1007/978-3-319-27146-0_28, doi:10.1007/978-3-319-27146-0_28.

[32] Lidiya Georgieva, Tatyana Dimitrova, e Nicola Angelov. RGB and HSV colour models in colour identification of digital traumas images. 2005. URL: http://ecet.ecs.uni-ruse.bg/cst05/Docs/cp/sV/V.12.pdf.

[33] Lih-Jen Kau e Tien-Lin Lee. An HSV Model-Based Approach for the Sharpening of Color Images. Em *2013 IEEE International Conference on Systems, Man, and Cybernetics*, páginas 150–155. IEEE, 10 2013. URL: http://ieeexplore.ieee.org/document/6721786/, doi:10.1109/SMC.2013.33.

[34] M J Mišić, Đ M Đurđević, e M V Tomašević. Evolution and trends in GPU computing. Em *MIPRO, 2012 Proceedings of the 35th International Convention*, páginas 289–294. IEEE, Piscataway, NJ, USA, 2012. URL: https://www.engineeringvillage.com/share/document.url?mid=inspec_10655dd138fda77797M57392061377553&database=ins.

[35] Irv Englander. *The Architecture of Computer hardware, System Software, and Networking*. 2009.

[36] Whitepaper Nvidia, Next Generation, e Cuda Compute. Fermi white paper. *ReVision*, 23(6):1–22, 2009. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, doi:10.1016/j.immuni.2005.11.006.

[37] Erik Lindholm, John Nickolls, Stuart Oberman, e John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 3 2008. URL: http://ieeexplore.ieee.org/document/4523358/, doi:10.1109/MM.2008.31.

[38] B. Ramakrishna Rau e Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1-2):9–50, 5 1993. URL: http://link.springer.com/10.1007/BF01205181, doi:10.1007/BF01205181.

[39] Karl Pauwels, Matteo Tomasi, Javier Diaz Alonso, Eduardo Ros, e Marc M. Van Hulle. A Comparison of FPGA and GPU for Real-Time Phase-Based Optical Flow, Stereo, and Local Image Features. *IEEE Transactions on Computers*, 61(7):999–1012, 7 2012. URL: http://ieeexplore.ieee.org/document/5936059/, doi:10.1109/TC.2011.120.

[40] Hongxing Wei, Zhenzhou Shao, Zhen Huang, Renhai Chen, Yong Guan, Jindong Tan, e Zili Shao. RT-ROS: A real-time ROS architecture on multi-core processors. *Future Generation Computer Systems*, 56:171–178, 3 2016. URL: https://www.sciencedirect.com/science/article/pii/S0167739X15001831, doi:10.1016/J.FUTURE.2015.05.008.

[41] Jorge Miguel Ferreira da Silva Mendes. Perceção visual semântica de uma vinha para auxílio à navegação de robôs. Tese de mestrado, UTAD, 8 2017. URL: https://repositorio.utad.pt/handle/10348/7939{#}.

[42] Chih-Chung Chang e Chih-Jen Lin. LIBSVM: A Library for Support Vector Machines. URL: https://www.csie.ntu.edu.tw/{~}cjlin//papers/libsvm.pdf.

[43] Qi Li, Raied Salman, Erik Test, Robert Strack, e Vojislav Kecman. GPUSVM: a comprehensive CUDA based support vector machine package. *Open Computer Science*, 1(4):387–405, 1 2011. URL: http://www.degruyter.com/view/j/comp.2011.1.issue-4/s13537-011-0028-7/s13537-011-0028-7.xml, doi:10.2478/s13537-011-0028-7.

[44] Jun Han e Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. páginas 195–201. Springer, Berlin, Heidelberg, 1995. URL: http://link.springer.com/10.1007/3-540-59497-3{_}175.

[45] Ting-Fan Wu, Chih-Jen Lin, e Ruby C Weng. Probability Estimates for Multi-class Classification by Pairwise Coupling. *Journal of Machine Learning Research*, 5:975–1005, 2004. URL: https://www.csie.ntu.edu.tw/{~}cjlin/papers/svmprob/svmprob.pdf.

[46] Naznin Fauzia, Louis-Noel Pouchet, e P. Sadayappan. Characterizing and enhancing global memory data coalescing on GPUs. Em *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, páginas 12–22. IEEE, 2 2015. URL: http://ieeexplore.ieee.org/document/7054183/, doi:10.1109/CGO.2015.7054183.

[47] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, e S. A. Jarvis. Achieving Portability and Performance through OpenACC. Em *2014 First Workshop on Accelerator Programming using Directives*, páginas 19–26. IEEE, 11 2014. URL: http://ieeexplore.ieee.org/document/7081674/, doi:10.1109/WACCPD.2014.10.

[48] Daniel Barba, Arturo Gonzalez-Escribano, e Diego R. Llanos. Comparative Analysis of OpenACC Compilers. páginas 91–104. Springer, Cham, 12 2016. URL: http://link.springer.com/10.1007/978-3-319-49956-7{_}7, doi:10.1007/978-3-319-49956-7_7.